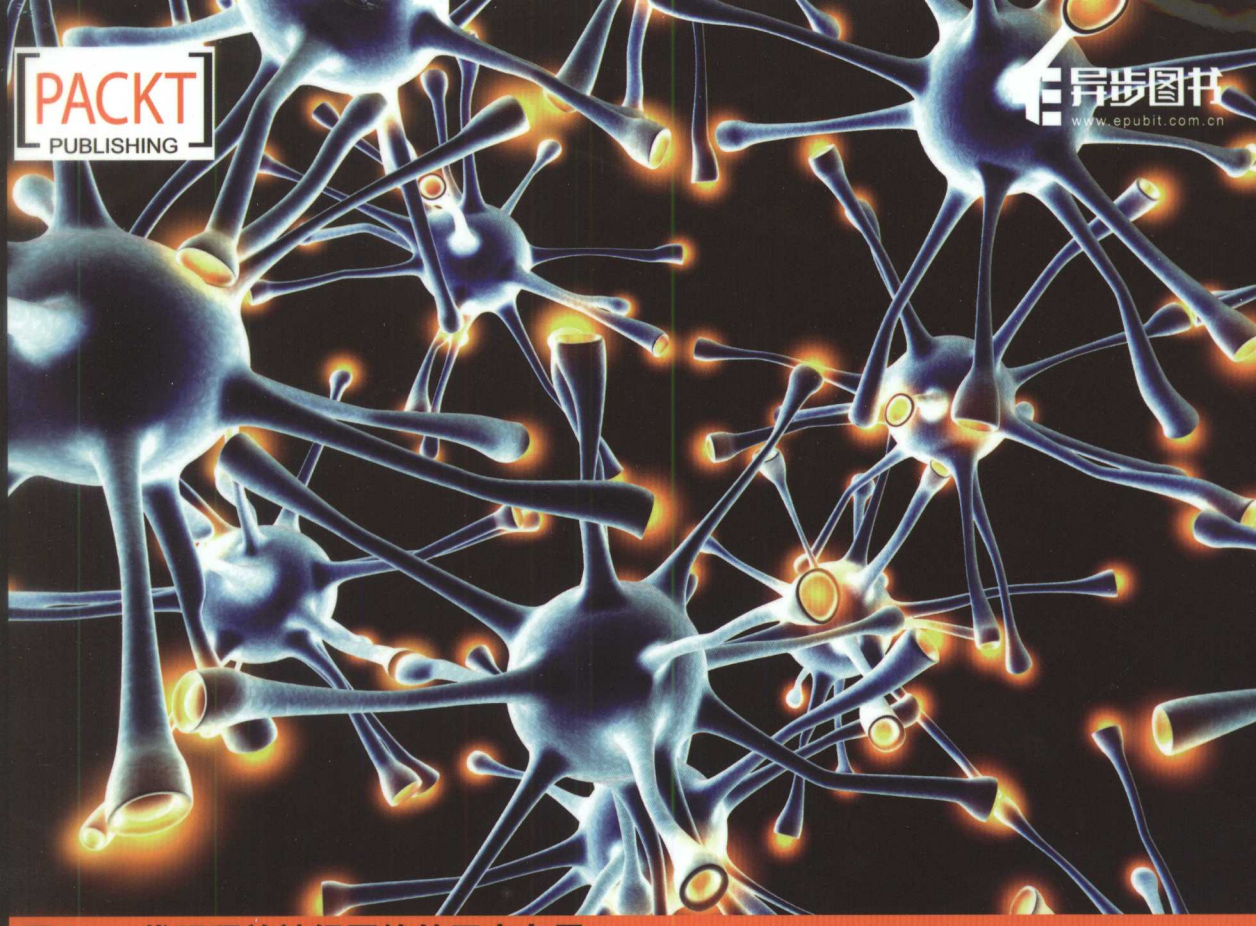


PACKT PUBLISHING

异步图书 www.epubit.com.cn

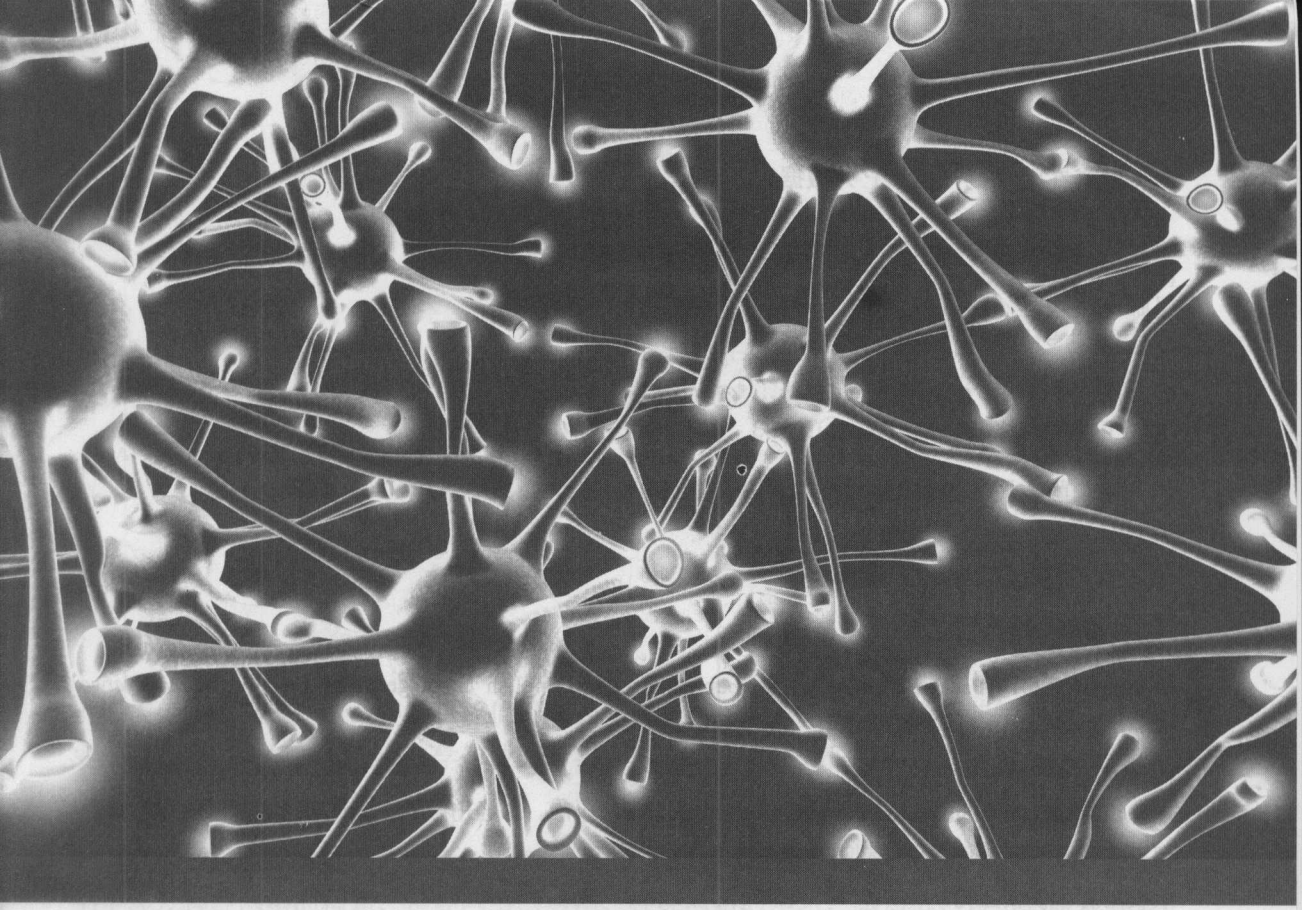
用Java代码释放神经网络的无穷力量

神经网络算法与实现
——基于Java语言

Neural Network Programming with Java

[巴西] Fábio M. Soares　Alan M.F. Souza　著
范东来　封强　译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS



神经网络算法与实现
——基于Java语言

[巴西] Fábio M. Soares　Alan M.F. Souza　著
范东来　封强　译

人民邮电出版社
北京

图书在版编目（CIP）数据

神经网络算法与实现 : 基于Java语言 / (巴西) 法比奥, (巴西) 艾伦著 ; 范东来, 封强译. -- 北京 : 人民邮电出版社, 2017.9
ISBN 978-7-115-46093-6

Ⅰ. ①神… Ⅱ. ①法… ②艾… ③范… ④封… Ⅲ. ①人工神经网络－算法 Ⅳ. ①TP183

中国版本图书馆CIP数据核字(2017)第156750号

版权声明

◆ 著　[巴西] Fábio M. Soares　Alan M.F. Souza
译　范东来　封　强
责任编辑　胡俊英
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：800×1000　1/16
印张：13
字数：250 千字　2017 年 9 月第 1 版
印数：1－2 400 册　2017 年 9 月河北第 1 次印刷

著作权合同登记号　图字：01-2016-8082 号

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

人工神经网络是由众多连接权值可调的神经元连接而成，具有大规模并行处理、分布式信息存储、良好的自组织自学习能力等特点，能够完成模式识别、机器学习以及预测趋势等任务。

本书通过 9 章内容，并结合 Java 编程语言，由浅入深地介绍了神经网络算法的应用。书中涉及神经网络的构建、神经网络的结构、神经网络的学习、感知机、自组织映射等核心概念，并将天气预测、疾病诊断、客户特征聚类、模式识别、神经网络优化与自适应等经典案例囊括其中。本书在附录中详细地指导读者进行开发环境的配置，帮助读者更加顺利地进行程序开发。

本书非常适合对神经网络技术感兴趣的开发人员和业余读者阅读，读者无需具备 Java 编程知识，也无需提前了解神经网络的相关概念。本书将从零开始为读者进行由浅入深地讲解。

作者简介

Fábio M. Soares 拥有帕拉联邦大学（Universidade Federal do Pará，UFPA）的计算机应用专业硕士学位，目前是该所大学的在读博士生。他从 2004 年开始就一直在设计神经网络解决方案，在电信、化学过程建模等多个领域开发了神经网络技术的应用，他的研究主题涉及数据驱动建模的监督学习。

他也是一名个体经营者，为巴西北部的一些中小型公司提供 IT 基础设施管理和数据库管理等服务。在过去，他曾为大公司工作，如 Albras（世界上最重要的铝冶炼厂之一）和 Eletronorte（巴西的一个大型电源供应商）。他也有当讲师的经历，曾在亚马逊联邦农业大学（Federal Rural University）和卡斯塔尼亚尔的一个学院授课，两所学校都在帕拉州，所教的学科涉及编程和人工智能。

他出版了许多作品，其中许多都有英文版，所有作品都是关于针对某些问题的人工智能技术。他在众多权威会议上发表了一系列学术文章，如 TMS（矿物金属和材料学会）、轻金属学会和智能数据工程、自动学习学会等学术会议。他还为 Intech 写过两章内容。

我特别庆幸自己能获得丰富的神经网络方面的知识，而我也只是单纯地喜欢做研究。特别感谢我的家人，我的父亲 Josafá 和母亲 Maria Alice，希望他们能因为这本书而感到非常自豪，还有我的兄弟 Flávio、我的姨妈 Maria Irenice，以及在我学习期间总是以某种方式支持我的所有亲戚。我还要感谢 Roberto Limão 教授的支持，他邀请我参与了许多关于人工智能和神经网络的项目。此外，特别感谢我的合作伙伴和来自 Exodus Sistemas 的前合作伙伴，他们帮助我应对编程和 IT 基础设施方面的各种挑战。最后，我要感谢我的朋友 Alan Souza，他邀请我成为本书的作者，并一起完成了本书。

Alan M.F. Souza 是来自亚马逊高级研究所（Instituto de Estudos Superiores da Amazônia，IESAM）的计算机工程师。他拥有软件项目管理的研究生学位以及帕拉联邦大学（Universidade Federal do Pará，UFPA）的工业过程（计算机应用）硕士学位。自 2009 年以来，他一直从事神经网络方面的工作，并从 2006 年开始与巴西的 IT 公司合作进行 Java、PHP、SQL 和其他编程语言的开发。他热衷于编程和计算智能。目前，他是亚马逊大学（Universidade da Amazônia，UNAMA）的教授和帕拉联邦大学的在读博士生。

在我小的时候，就想写一本书。所以，本书是我一个梦想的实现，也是我努力工作的结果。我要感谢这个机会，也要感谢我的父亲 Célio、我的母亲 Socorro、我的妹妹 Alyne 和我了不起的妻子 Tayná。他们能够理解我不能时常陪伴他们，还经常为我担心。我感谢所有家庭成员和朋友在困难时期一直支持我，并期待我的成功。我要感谢所有出现在我生命中的教授，特别是 Roberto Limão 教授，他第一个传授给我神经网络的概念。我还要向 Fábio Soares 表示感谢，感谢这次伟大的合作和这段伟大的友谊。最后，感谢 Packt 出版社这个不知疲倦的团队，感谢他们在整个出版过程中给予的帮助。

译者简介

范东来

北京航空航天大学硕士，BBD（数联铭品）大数据技术部负责人，大数据平台架构师。技术图书作者和译者，著有《Hadoop 海量数据处理》，译有《解读 NoSQL》《NoSQL 权威指南》。研究方向：大规模并行图挖掘、去中心化应用。

封强

毕业于北京化工大学计算机科学系、法国图尔工程师学院，获法国工程师学位，目前是 BBD（数联铭品）大数据技术部工程师。研究方向：数据仓库、商业智能。

审稿人简介

Saeed Afzal，也被称为 Smac Afzal，是定居在巴基斯坦的专业软件工程师和技术爱好者。他专注于解决方案架构和可扩展的高性能应用的实现。

他热衷于为不同 Web 业务需求提供自动化解决方案。他目前的研究和工作主要是对下一代 Web 开发框架的实现，这会减少开发的时间和成本，并在默认情况下提供具有许多必要的强大功能的高效网站。他希望在 2016 年推出这个技术。

他还参与了 Packt 出版社出版的 *Cloud Bees Development* 一书的审校。

你可以在 `http://sirsmac.com` 上找到更多关于他技能和经验的信息，也可以通过 `sirsmac@gmail.com` 联系他。

我想感谢我的父母和妻子，感谢 Zara Saeed 博士，感谢他们给予的所有鼓励。

前言

程序员的生活就是一条持续的、永无止境的学习之路。程序员总是面临新技术或新方法的挑战。一般来说，在生活中，虽然我们习惯了做重复的事情，但总是会去接受并学习新的东西。学习的过程是最有趣的科学话题之一，人们做了许多尝试来描述或重现人类的学习过程。

面对新内容并且掌握新内容这样的挑战指导着本书的写作。虽然“神经网络”这个名字可能看起来很奇怪，甚至给人以本书是关于神经学的一种感觉，但我们努力把焦点集中在你决定购买本书的原因上，来简化这些细微差别。我们打算建立一个框架，以显示神经网络实际上是简单易懂的，并且保证读者无需预先了解这个题目，也能充分理解本书提出的概念。

所以，我们鼓励你全面地探究本书的内容，在面对重大问题时，要看到神经网络的强大力量，但要始终保持一个初学者的视角。本书中讲述的每一个概念都是用简单的技术性语言来解释的。本书的宗旨是让你可以使用简单语言来编写智能应用程序。

最后，感谢所有直接或间接对本书做出贡献的人，是他们从一开始就支持我们，感谢帕拉联邦大学、数据和组件提供者巴西气象学会（Brazilian Institute of Meteorology，INMET）、Proben1 和 JFreeCharts。特别感谢我们的顾问 Roberto Limão 教授，他给我们介绍神经网络这个主题，我们合作在这个领域发表了许多论文。还要感谢参考文献中引用的几位作者所做的工作，他们拓宽了我们对神经网络的视野，并启发我们深入思考如何指导读者将神经网络应用于 Java 编程。

我们希望你有一个非常愉快的阅读体验，鼓励你下载源代码，并遵循本书中引用的代码示例。

本书涵盖的内容

第 1 章，初识神经网络，这是神经网络基础知识及其用途介绍。你将了解到本书所涉及的基本概念以及对 Java 编程语言的简要回顾。在所有随后的小节中，还包括神经网络的 Java 代码实现。

第 2 章，神经网络是如何学习的，涵盖神经网络的学习过程，并展示如何使用数据来达到这一目的。这里介绍了学习算法的完整结构和设计。

第 3 章，运用感知机，介绍感知机的使用。感知机是最常用的神经网络架构之一，本书提出一个包含神经元层的神经网络结构，并展示它们在基础问题上如何通过数据来学习。

第 4 章，自组织映射，展示了一个无监督的神经网络架构（自组织映射），它被应用于在记录中寻找模式或聚类。

第 5 章，天气预测，这是第一个关于实践的章节，展示了神经网络在预测，即天气数据预测方面的一个有趣的应用。

第 6 章，疾病诊断分类，涵盖另一个实用的且神经网络非常擅长的分类任务。在本章中，你将看到一个非常有说服力而且有趣的疾病诊断应用。

第 7 章，客户特征聚类，介绍神经网络如何能够在数据中找出模型，包含一个常见的应用开发，涉及对具有相同购买属性的客户进行分组。

第 8 章，模式识别（OCR 案例），一谈到有趣且惊人的模式识别功能，肯定包括光学字符识别，本章探讨如何使用 Java 语言的神经网络来完成此任务。

第 9 章，神经网络优化与自适应，介绍了关于如何优化和增加神经网络的适应性，从而增强神经网络的能力。

阅读本书所需的开发工具

你需要 Netbeans (`www.netbeans.org`) 或 Eclipse (`www.eclipse.org`)，它们都是免费的且可以在其官方网站下载。

本书适合的读者

本书面向神经网络技术的专业开发人员和业余爱好者，读者无需具备 Java 编程知识。阅读本书不需要有神经网络的相关知识，本书将从头开始教。如果读者熟悉神经网络或其他机器学习技术，即使缺乏 Java 经验，本书也能提升其开发能力，帮助读者开发出实用的应用程序。当然，如果读者知道基本的编程概念，将会从这本书中获益良多，但无需以前的经验。

规范

本书采用不同的文本样式区分不同种类的信息。这里列举部分示例并对其含义作出解释。

文本中的代码、数据库表名、文件夹名称、字符串、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 句柄如下所示："在 Java 项目中，这些值的计算是通过 Classification 类完成的。"

代码块如下所示：

```
Data cardDataInput = new Data("data", "card_inputs_training.csv");
Data cardDataInputTestRNA = new Data("data", "card_inputs_test.csv");
Data cardDataOutputTestRNA = new Data("data", "card_output_test.csv");
```

新的术语和**重要的文字**以**粗体**展示：

警告信息或重要注释出现在这样的方框里。

温馨提示和小技巧出现在这样的框里。

读者反馈

我们非常欢迎来自读者的反馈。让我们知道你对本书的看法，喜欢什么或不喜欢什么。读者的反馈很重要，因为它有助于我们开发出真正有价值的图书内容。

如果读者要向我们发送一般性反馈，只需发送电子邮件至 feedback@packtpub.com，并在邮件主题中注明图书的标题。

如果你对于一个主题有非常专业的知识，并且对编写或贡献一本书感兴趣，请参阅我们的作者指南 www.packtpub.com/authors。

读者支持

现在，你已经是一名 Packt 图书的尊享者，我们有一些东西可以帮助你获取更多的信息。

下载示例代码

你可以访问 http:// www. packtpub.com，在账户下载所有已购买的 Packt 图书的示例代码文件。如果你是从其他地方购买本书，则可以访问 http://www.packtpub.com/support 并注册，我们将以电子邮件的形式直接发送给你。

勘误

虽然我们已经很谨慎，以确保内容的准确性，但是错误仍在所难免。如果你在我们的任何一本书中发现了错误，无论是文本或代码中的错误，如果你能向我们报告这些错误，我们将不胜感激。通过这样的方式，可以解决其他读者的困惑，并帮助我们改进本书的后续版本。如果你发现任何勘误，请通过访问 http: //www.packtpub.com / submit-errata，选择要报告的图书，单击勘误提交表单链接，并输入勘误的详细信息。一旦你的勘误被验证成功，提交将被接受，勘误表将被上传到我们的网站或添加到该标题的勘误表下任何一个现有的勘误表里。

关于盗版行为

在互联网上，盗版是所有媒体正在面对的问题。在 Packt，我们非常重视版权保护和版权许可。如果你在互联网上发现任何形式的 Packt 图书的非法副本，请立即向我们提供网址或网站名称，以便我们可以采取补救措施。

请通过 copyright@packtpub.com 向我们提供可疑盗版材料的链接。

感谢你帮助我们保护作者，以及保护我们为你带来有价值内容的能力。

疑问

如果你对本书的任何方面有问题，可以通过 questions@packtpub.com 与我们联系，我们将尽最大努力为你解决问题。

目录

第 1 章
初识神经网络

本章介绍神经网络及其用途，主要讲解神经网络的基本概念，为后续章节打下基础。其主要内容包含以下几个部分：

- 人工神经元
- 权值和偏置
- 激活函数
- 神经元层
- 用 Java 实现神经网络

1.1 探索神经网络

首先，“神经网络”这个词会在我们脑海里创建出一种模拟人脑的景象，特别是刚刚接触到神经网络的人可能产生这样的想法。实际上，这也是正确的，大脑就是一个巨大的自然神经网络。然而，人工神经网络又是什么呢？其实“人工”是相对于自然神经网络中的“自然”而言的，一提到“人工”一词，首先映入脑海的往往是一张人工大脑或机器人的图片。我们受人脑结构的启发创造了与人脑相似的神经网络结构，所以这也可以被叫作人工智能。读到这里，之前对人工神经网络没有什么经验的读者可能认为这本书是教大家怎么创造智能系统的，例如人造大脑就可以用 Java 代码模拟人类大脑思考的过程，不是吗？当然，我们不会涉及像黑客帝国里人工智能机器人那样的产物，但是，我们会谈到神经网络可以实现很多不可思议的功能。充分利用 Java 编程语言框架的优势，为读者提供定义和创建神经网络结构的完整 Java 代码。

1.2 为什么要用人工神经网络

在不清楚人工神经网络的起源，包括一些人工神经网络的专业术语时，我们不宜马上展开讨论。在本书中，我们认为神经网络（neural network，NN）和人工神经网络（artificial neural network，ANN）是同一个概念，尽管神经网络的概念更加宽泛，它还包括自然神经网络。那么，到底什么是ANN？让我们来简单探索这个术语的历史。

20世纪40年代，神经生理学家Warren McCulloch和数学家Walter Pits设计了第一个包含神经系统学基础和数学操作的人工神经元的数学实现模型。在那时，很多研究的方向是弄清楚人类大脑的结构和它是否可以被模拟以及如何模拟，但是这些研究都只停留在神经系统学科领域内。McCulloch和Pits的想法是非常创新的，因为他们引入了数学的元素。另外，想想我们的大脑由数十亿计的神经元组成，每个神经元又和其他数百万神经元相连接，最终得到数万亿计的神经元连接，这是一个多么巨大的神经网络。然而，每个神经元单元是非常简单的，仅仅是作为一个能够汇总求和以及传递信号的处理器。

基于此，McCulloch和Pits设计了一个简单的单个神经元模型，最初用于模拟人类的视觉。那时候的计算器和计算机还是非常少的，但是它们善于快速处理数学操作。另外，即使是现在，如果不用一些特殊的编程框架，视觉处理和声音识别的编程也是很难的，这恰恰与计算机能快速处理函数和数学操作的事实相反。然而，如表1-1所示，人脑在视觉处理和声音识别方面比计算器和计算机更高效，这确实激励了很多科学家和研究人员从事这方面的研究。

因此，人们认为人工神经网络能够很好地完成诸如模式识别、机器学习，以及预测趋势等任务，就像一个专家可以根据基础知识进行工作一样。而与那些需要执行一系列步骤来完成一个既定目标的传统算法方法恰恰相反，人工神经网络反而有能力通过它高度互联的网络结构去学习怎样自己解决问题。

表1-1

人类能快速解决的任务	计算机能快速解决的任务
图片分类	复杂计算
声音识别	语法错误纠正
人脸识别	信号处理
通过经验预测事件	操作系统管理

1.3 神经网络的构造

可以说人工神经网络是受自然启发的结构，它和人脑有很多相似之处。图 1-1 展示了一个自然神经元结构，它由神经元细胞核、树突和轴突组成，轴突分出很多分支来与其他神经元的树突相连而形成突触。

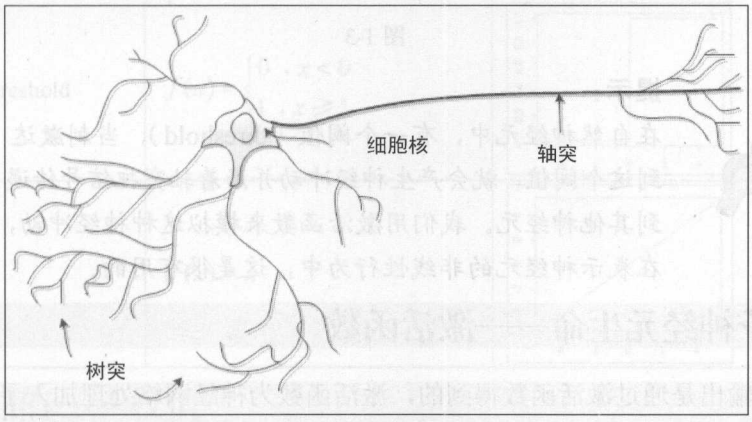


图 1-1

所以，人工神经元有相似的结构，它也包含一个核（处理单元）、多个树突（类似于输入）以及一个轴突（类似于输出），如图 1-2 所示。

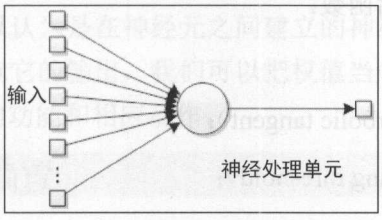


图 1-2

在所谓的神经网络中，神经元之间的连接类似于自然神经结构中的突触。

1.3.1 基础元素——人工神经元

自然神经元被证实为一个信号处理器，这是由于它可以在树突端接收微信号，根据信号的强度或者大小，会在轴突触发一个信号。我们可以认为神经元在输入端有一个信号接收器，在输出端有一个响应单元，它可以根据不同的强度和量级触发一个可以向前传递至其他神经元的信号。因此，我们可以定义一个人工神经元结构，如图 1-3 所示。

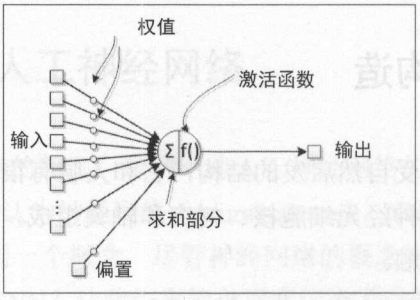


图 1-3

提示：

在自然神经元中，有一个阈值（threshold），当刺激达到这个阈值，就会产生神经冲动并沿着轴突把信号传递到其他神经元。我们用激活函数来模拟这种神经冲动，在表示神经元的非线性行为中，这是很有用的。

1.3.2 赋予神经元生命——激活函数

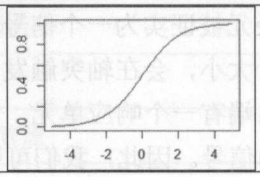
神经元的输出是通过激活函数得到的，激活函数为神经网络处理加入了非线性特征，由于自然神经元具有非线性行为，所以非线性特征是非常必要的。激活函数往往把输出信号限制在一定范围内，因此，激活函数常常是非线性函数，但是在一些特殊情况下，也可能是线性函数。

如下是 4 种最常用的激活函数：

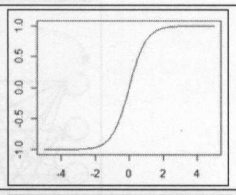
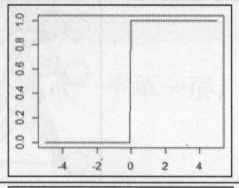
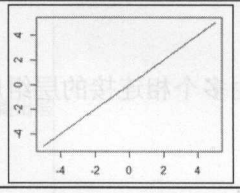
- S 函数（Sigmoid）；
- 双曲正切函数（Hyperbolic tangent）；
- 阈值函数（Hard limiting threshold）；
- 纯线性函数（Purely linear）。

表 1-2 展示的是与这些函数对应的公式及图像。

表 1-2

函数	公式	图像
Sigmoid	$f(x)=\dfrac{1}{1+e^{-x}}$	

续表

函数	公式	图像
Hyperbolic tangent	$f(x)=\dfrac{1-e^{-x}}{1+e^{-x}}$	
Hard limiting threshold	$f(x)=\begin{cases}0 & ,x<0\\1 & ,x\geqslant 1\end{cases}$	
Linear	$f(x)=x$	

1.3.3 基础值——权值

在神经网络中，权值代表着神经元之间的连接并且它可以放大或减小神经元信号，例如扩大信号从而改变信号。因此，通过改变神经网络信号，神经系统权值有能力影响神经元的输出信号，所以一个神经元的激活依赖于输入和相应的权值。倘若输入信号来自其他神经元或外界，那么权值可以认为是在神经元之间建立的神经网络连接。由于权值是神经网络的内部因素并且可以影响它的输出，我们可以把权值当成神经网络的知识，只要更改了权值，将会改变神经网络的功能和相应动作。

1.3.4 重要参数——偏置

人工神经元可以拥有一个独立的元素，它可以把外部信号添加到激活函数，这个独立的元素被称为偏置。

就像输入信号有一个对应的权值，偏置也拥有一个对应的权值，这个特性可以使神经网络知识表示成一个更纯粹的非线性系统。

1.3.5 神经网络组件——层

自然神经元以层的方式组织，每一层都有自己的处理方式，例如输入层接受外界直接刺激，输出层产生可以影响外界的神经冲动。在这些层之间，有很多隐藏层，意味着它们

不会和外界直接产生相互影响。在人工神经网络中，同一层的所有神经元共享相同的输入和激活函数，如图 1-4 所示。

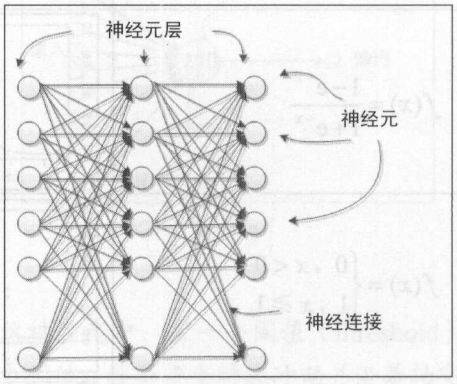


图 1-4

神经网络由多个相连接的层组成，形成多层网络。这些神经系统层可以分成以下 3 种基本类型：

- 输入层；
- 隐藏层；
- 输出层。

实际上，接受外界刺激的抽象神经系统层可以作为一个附加层，以此来增强神经网络对更加复杂的知识的表现力。

提示：

每个神经网络至少含有一个输入层/输出层，不管神经网络的层数是多少。在多层神经网络的情况下，在输入层和输出层之间的都被称为隐藏层。

1.4 神经网络结构

一般来说，神经网络含有不同的结构，这取决于神经元或者神经元层之间的连接方式。每一种神经网络结构都针对特定问题。神经网络可以用来处理很多问题，我们根据问题的不同特征来设计神经网络结构，可以更有效地处理这个问题。

主要有两类神经网络结构模型：

- 神经元连接

- 单层神经网络
- 多层神经网络

- 信号流
  - 前馈神经网络
  - 反馈神经网络

1.4.1 单层神经网络

在单层神经网络结构中，所有的神经元位于相同层，形成一个单一层，如图 1-5 所示。

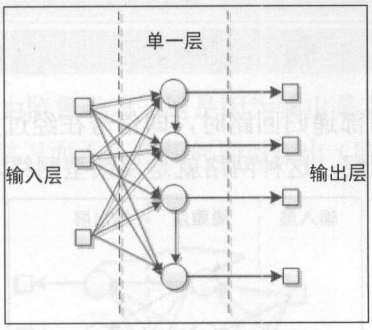


图 1-5

神经网络接收输入信号并将其传递给神经元，然后神经元产生输出信号。神经元之间的连接可以是重复的，也可以是不重复的。这种结构可用于单层感知机、自适应机、自组织映射、Elman 网络和 Hopfield 神经网络。

1.4.2 多层神经网络

对于多层神经网络，神经元被分成多个神经元层，每一层对应一个共享相同输入数据的并行神经元结构，如图 1-6 所示。

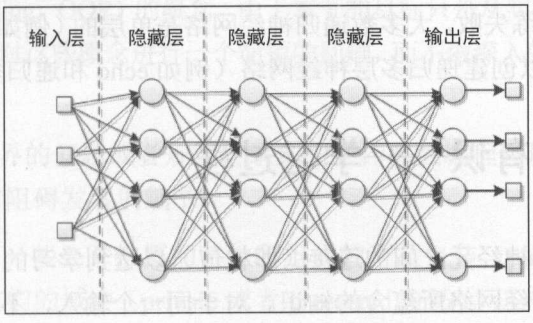


图 1-6

径向基函数和多层感知机是多层神经网络结构典型的实例，这种网络结构非常适合用于逼近函数所表示的真实数据。此外，由于多层神经网络结构含有多个处理层，它比较适合于训练一组非线性数据，然后可以区分数据或更简单地确认数据复制或数据识别的知识。

1.4.3 前馈神经网络

神经网络中的信号流可以是单向的，也可以是循环的。在第一种情况下，我们把这种神经网络结构称为前馈神经网络，从输入信号进入输入层到信号被处理后，信号都是向前传递到下一层的，如图 1-6 所示。多层感知机和径向基函数同样是前馈神经网络很好的实例。

1.4.4 反馈神经网络

当神经网络中出现一些内部递归回路时，即信号在经过处理之后又被传回到接收和处理过该信号的神经元或神经元层，这种网络就是反馈型网络，如图 1-7 所示。

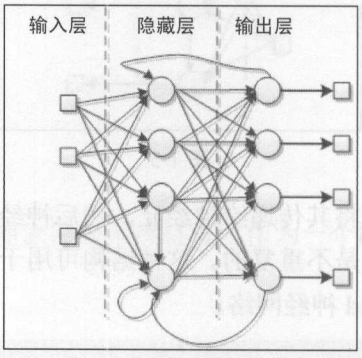


图 1-7

在神经网络中添加递归回路的理由是引入动态行为，特别是在处理涉及时间序列或模式识别的时候，这就需要一个内部存储来增强学习过程。但是，这种神经网络的训练是特别艰难的，最终可能训练失败。大多数递归神经网络是单层的（例如 Elman 网络和 Hopfield 神经网络），但是也可以创建递归多层神经网络（例如 echo 和递归多层感知机网络）。

1.5 从无知到有识——学习过程

神经网络通过调整神经元之间的连接（即权值）以达到学习的目的。就像在 1.3 节中提到的，权值代表了神经网络所蕴含的知识。对于同一个输入，不同的权值将会产生不同的结果。神经网络可以根据某种学习规律来调整其权值，以提升预测结果质量。学习过程

的一般模式如图 1-8 所示。

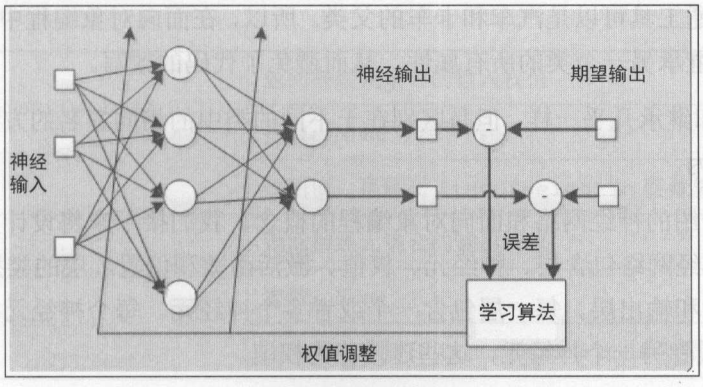


图 1-8

图 1-8 所描绘的过程被称为监督学习，这是因为输出是某种预期的值，但是神经网络也可以仅通过其输入数据进行学习而不依赖任何期望输出（监督）。在第 2 章中，我们将深入了解神经网络的学习过程。

1.6 实践神经网络

本书将会用 Java 编程语言来实现一个神经网络的整个过程。Java 于 20 世纪 90 年代由太阳微系统公司（Sun Microsystems）的工程师发明，是一种面向对象的编程语言，其所有权于 2010 年被甲骨文公司（Oracle）获得。如今，Java 运行在许多设备中，这些设备已成为我们日常生活的一部分。

在 Java 等面向对象语言中，我们同类和对象打交道。类是真实世界中某些事物的模板，而对象则是这种模板对应的实例，有点像汽车（代表所有以及各种车的类）和我的汽车（代表某辆特定的车）。Java 类往往由属性和方法（或者函数）构成，它包含了面向对象编程（object-oriented programming，OOP）的概念。由于本书的目标只是从实践的角度来设计和创建神经网络，所以我们只会对这些概念进行一个简短的回顾，而不做深入的探究。在这个过程中，需要了解 4 个相关概念。

- **抽象**：将真实世界的问题或者规律转录到某个计算机编程领域，只考虑其相关特征而忽略那些经常阻碍发展的细节。
- **封装**：类似于由一些公开披露的特性所组成的产品包装（public 方法），然而其他方法被隐藏在它们的域中（private 或者 protected 方法），这样可以避免误用信息或者滥用信息。

- **继承**：在真实世界中，多个类的对象可以以一种分层的方式共用属性和方法；例如，某种交通工具可以是汽车和卡车的父类。所以，在面向对象编程中，这个概念允许一个类继承另一个类的所有属性，从而避免了代码的改写。
- **多态**：和继承几乎一样，但是区别在于不同的类中的相同签名的方法可以呈现出不同的行为。

根据本章介绍的神经网络和面向对象编程的概念，我们接下来将设计第一个实现神经网络的类集。神经网络包含层、神经元、权值、激活函数和偏置，层的类型主要有 3 种：输入层、隐藏层和输出层。每一层包含一个或者多个神经元。每个神经元都被连接到某个神经输入/输出或者另一个神经元，这些连接称为权值。

要特别强调一点，神经网络可以有许多隐藏层，也可以一个都没有，每一层的神经元数量也会变化。但是，输入和输出层的神经元数量分别与神经输入和神经输出的数量相等。

接下来，让我们开始实现。首先，我们来定义 6 个类，细节如表 1-3 所示。

表 1-3

类名：Neuron	
属性	
`private ArrayList<Double> listOfWeightIn`	一个实数 `ArrayList` 变量，代表输入权值的集合
`private ArrayList<Double> listOfWeightOut`	一个实数 `ArrayList` 变量，代表输出权值的集合
方法	
`public double initNeuron()`	用伪随机实数初始化 `listOfWeightIn` 和 `listOfWeightOut`
	参数：None
	返回值：一个伪随机实数
`public void setListOfWeightIn(ArrayList<Double> listOfWeightIn)`	用一个实数集合设置 `listOfWeightIn` 的函数
	参数：将被存储在类对象里的实数列表
	返回值：None
`public void setListOfWeightOut(ArrayList<Double> listOfWeightOut)`	用一个实数集合设置 `listOfWeightOut` 的函数
	参数：将被存储在类对象里的实数列表
	返回值：None

续表

方法	
public ArrayList<Double> getListOfWeightIn()	返回神经元集合的输入权值
	参数：None
	返回值：存储在 listOfWeightIn 变量里的实数集合
public ArrayList<Double> getListOfWeightOut()	返回神经元集合的输出权值
	参数：None
	返回值：存储在 listOfWeightOut 变量里的实数集合
Java 实现类：Neuron.java 文件	
类名：Layer	
注意：该类为抽象类且不能被实例化	
属性	
private ArrayList<Neuron> listOfNeurons	一个 ArrayList 变量，元素类型为 Neuron 类
private int numberOfNeuronsInLayer	用来存储层的神经元数量的整数
方法	
public ArrayList<Neuron> getListOfNeurons()	返回层的神经元集合
	参数：None
	返回值：一个 ArrayList 变量，元素类型为 Neuron 类
public void setListOfNeurons(ArrayList<Neuron> listOfNeurons)	用一个 Neuron 集合设置 listOfNeurons 的函数
	参数：将被存储的 Neuron 集合
	返回值：None
public int getNumberOfNeuronsInLayer()	返回层的神经元的数量
	参数：None
	返回值：层的神经元数量

续表

方法	
public void setNumberOfNeuronsInLayer(int numberOfNeuronsInLayer)	设置层的神经元数量
	参数：层的神经元数量
	返回值：None
Java 实现类：Layer.java 文件	
类名：InputLayer	
注意：该类从 Layer 类继承其属性和方法	
属性	
None	
方法	
public initLayer(InputLayer inputLayer)	用伪随机数初始化输入层
	参数：一个 InputLayer 类的对象
	返回值：None
public void printLayer(InputLayer inputLayer)	打印层的输入权值
	参数：一个 InputLayer 类的对象
	返回值：None
Java 实现类：InputLayer.java 文件	
类名：HiddenLayer	
注意：该类从 Layer 类继承其属性和方法	
属性	
None	
方法	
public ArrayList<HiddenLayer> initLayer(HiddenLayer hiddenLayer, ArrayList<HiddenLayer> listOfHiddenLayer, InputLayer inputLayer, OutputLayer outputLayer)	用伪随机实数初始化隐藏层
	参数：一个 HiddenLayer 类的对象，一个元素类型为 HiddenLayer 的集合，一个 Inputlayer 类的对象，一个 OutputLayer 类的对象
	返回值：None

续表

方法	
public void printLayer(ArrayList<HiddenLayer> listOfHiddenLayer)	打印层的权重
	参数：一个元素对象为 HiddenLayer 类型的集合
	返回值：None
Java 实现类：HiddenLayer.java 文件	
类名：OutputLayer	
注意：该类从 Layer 类继承其属性和方法	
属性	
None	
方法	
public OutputLayer initLayer (OutputLayer outputLayer)	用伪随机数初始化输出层
	参数：一个 OutputLayer 类的对象
	返回值：None
public void printLayer(OutputLayer outputLayer)	打印层的权值
	参数：一个 OutputLayer 类的对象
	返回值：None
Java 实现类：OutputLayer.java 文件	
类名：NeuralNet	
注意：神经网络拓扑结构在该类中是固定的（输出层有两个神经元，两个隐藏层，每层分别有 3 个神经元，输出层有一个神经元） 提示：这是第一个版本	
属性	
private InputLayer inputLayer;	一个 InputLayer 类的对象
private HiddenLayer hiddenLayer;	一个 HiddenLayer 类的对象
Private ArrayList<HiddenLayer> listOfHiddenLayer;	一个元素类型为 HiddenLayer 的 ArrayList 变量。可能会存在不止一个隐藏层
private OutputLayer outputLayer;	一个 OutputLayer 类的对象
private int numberOfHiddenLayers;	存储隐藏层的层数量的整数

续表

方法	
public void initNet()	将神经网络作为一个整体初始化。层会被构建，神经元的权值的每个集合会被随机构建
	参数：None
	返回值：None
public void printNet()	将神经网络作为一个整体打印。每一层的每个输出和输入权值都会被展示
	参数：None
	返回值：None
Java 实现类：NeuralNet.java	

OOP 语言的一个优势是易于在统一建模语言（UML）中用文档来表现程序。UML 类图以一种简单而直观的方法展现了类、属性、方法和不同的类之间的关系，这样就能帮助程序员及其利益相关者将整个项目作为一个整体来理解。图 1-9 展现了该项目的最初版本类图。

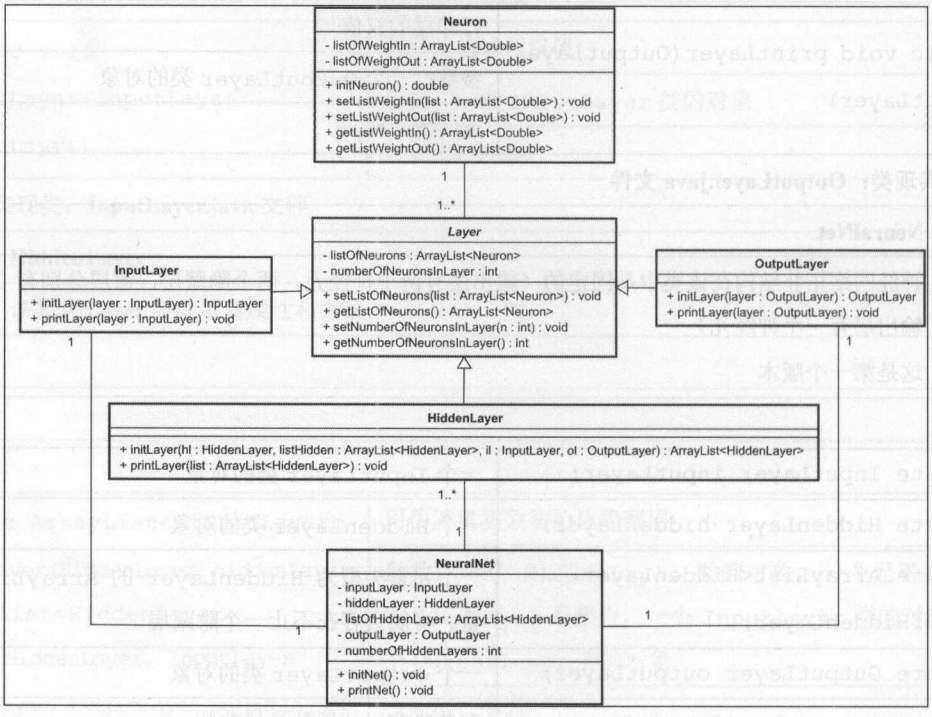


图 1-9

现在，让我们来应用这些类得到一些结果。下面展示的代码有一个测试类、一个 main 方法和一个名为 n 的 NeuralNet 类的对象。当这个方法被调用（通过类执行时），它会从对象 n 中调用 initNet()和 printNet()方法，生成如图 1-10 所示的结果。它表示了一个神经网络，输入层有两个神经元，隐藏层有 3 个神经元，输出层有一个神经元：

```
public class NeuralNetTest {
  public static void main(String[] args) {
    NeuralNet n = new NeuralNet();
    n.initNet();
    n.printNet();
  }
}
```

需要记住的是每次运行代码，会生成相应的新的伪随机权值。所以，当你运行这段代码时，其他值将会出现在控制台中，如图 1-10 所示。

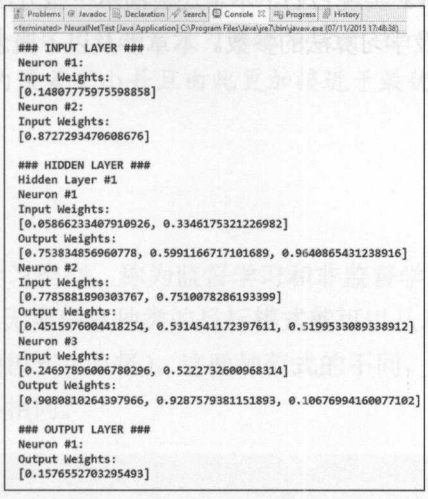


图 1-10

1.7 小结

在本章中，我们了解了神经网络的定义、用途和基本概念。我们还了解了用 Java 编程语言实现的一个非常基础的神经网络，通过编码神经网络的每个元素，在实践中应用了神经网络理论。在进入高级概念之前，了解这些基本概念很重要。这同样适用于 Java 实现的代码。在下一章中，我们将深入研究神经网络的学习过程，并且以简单的实例来探索不同的学习类型。

第 2 章
神经网络是如何学习的

本章将展示神经网络从数据中获取知识而执行的学习过程。我们将介绍训练、测试和验证等概念，并且展示如何用 Java 实现它们。本章还会演示一些方法，这些方法用来评估某个神经网络在学习方面的表现以及学习算法的参数。本章涉及以下概念：

- 学习过程
- 学习算法
- 学习类型
  - 监督
  - 无监督
- 训练、测试和验证
- 误差测量
- 泛化

2.1 神经网络的学习能力

神经网络真正令人惊叹的是其从周围环境中学习的能力，就像那些有天赋的人能做的一样。我们人类通过观察和重复练习来体验学习过程，直到某些任务或者概念被完全掌握。从生理学的角度来说，在人脑中的学习过程是对节点（神经元）之间的神经连接进行重新设置，这个过程导致了一个新思维定式的产生。

神经网络的连接特性使得神经网络将学习过程分布到整个结构，这个特性使得这种结构可以足够灵活来学习各种各样的知识。与那些只能执行预先设定好模式的任务的数字计

算机相反，神经系统能够根据预设的期望标准来提升和执行新任务。换句话说，神经网络不需要预先设定模式；它们能自己学习模式。

神经网络如何有助于解决问题

考虑到每个需要解决的任务可能有很多理论上可行的解，神经网络的学习过程旨在寻找一种能够生成满意结果的最优解。像人工神经网络这样的结构被鼓励使用，这是由于它们可以严格从输入刺激中获取任意类型的知识，也就是说，和数据相关的任务或问题。首先，人工神经网络会生成一个随机结果和误差，然后根据这个误差，其参数会进行调整。

提示：

我们可以认为人工神经网络的参数（权值）是解的一部分。假设单个解表示在解的多维空间中的一个点。每个解产生一个误差度量，它告诉我们这个解与最优解的差距。在每次迭代中，学习算法都会寻找一个解，这个解产生的误差更小并且由此更加接近于最优解。

2.2 学习范式

神经网络主要有两种学习类型，称为监督学习和非监督学习。人类思维的学习类型无外乎也是这两种方式。我们无需任意种类的目标模式就可以从观察中学习（无监督），或者遵循老师给我们展示的正确模式（监督）。这两种范式的不同，主要体现在目标模式的相关性上，对不同的问题也不尽相同。

2.2.1 监督学习

监督学习处理形如 X、Y 的键值对，其目标是通过函数 f 将 X 映射到 Y 上：f: $X \rightarrow Y$。Y 在这里就是监督者，是期望的目标输出，而 X 数据是独立于源的数据，它可以生成 Y 数据。这类似一位老师，正在教授某人将要被执行的特定任务，如图 2-1 所示。

图 2-1

该学习范式的一个特性是存在一个直接的误差参考，也就是将目标和实际结果相比较。神经网络的参数会被输入到一个代价函数，它可以量化期望和实际输出之间的误差。

提示：

在最优化问题中，代价函数只是一个需要最小化的度量标准。这意味着人们将寻找使代价函数为最低可能值的参数。代价函数将在本章后面进行更详细的介绍。

监督学习非常适合那些已经提供了某种模式、某个需要达到目标的任务。例如后面的一些案例：图像分类、语音识别、函数逼近和预测。注意，一些先验知识，如独立的输入值(X)和分类标注的输出值(Y)需要提供给神经网络。有标注的输出值是监督学习的必要条件。

2.2.2 无监督学习

如图 2-2 所示，在无监督学习中，我们处理那些没有任何标注或者分类的数据；换句话说，神经结构会试图举一反三并只考虑从输入数据 X 中提取知识。

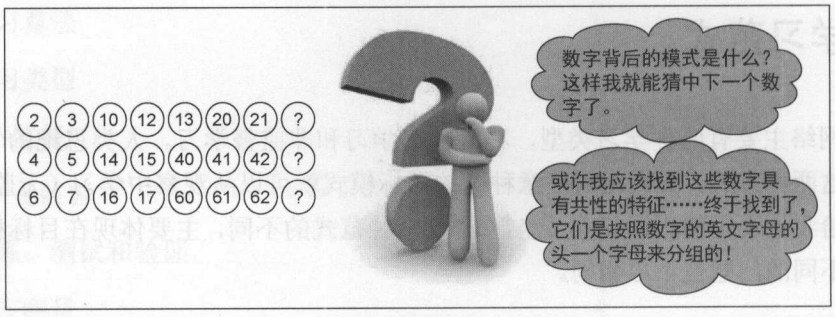


图 2-2

这类似于自我学习，就像某人通过自己的经验和一些支持标准来学习。在无监督学习里，我们不需要定义一个期望的模式来应用到观测值里，但是神经结构能够在没有任何监督的情况下自己生成一个模式。

提示：

这里，代价函数扮演了一个重要的角色。它将在很大程度上影响所有神经属性以及输入数据之间的关系。

无监督学习可以应用到如下场景：聚类、数据压缩、统计建模和语言建模。这种学习范式在第 4 章中会详细讨论。

2.3 系统结构——学习算法

我们已经从理论上定义了学习过程和它的工作原理。接下来必须继续深入探讨学习算法本身的数学逻辑。学习算法是驱动神经网络学习过程的手段，它在很大程度上由神经网络架构来决定。从数学的观点上说，它旨在寻找一个能使代价函数 $C(X,[Y])$尽可能小的 W 权值的最优解。

通常，学习算法的执行流程如图 2-3 所示。

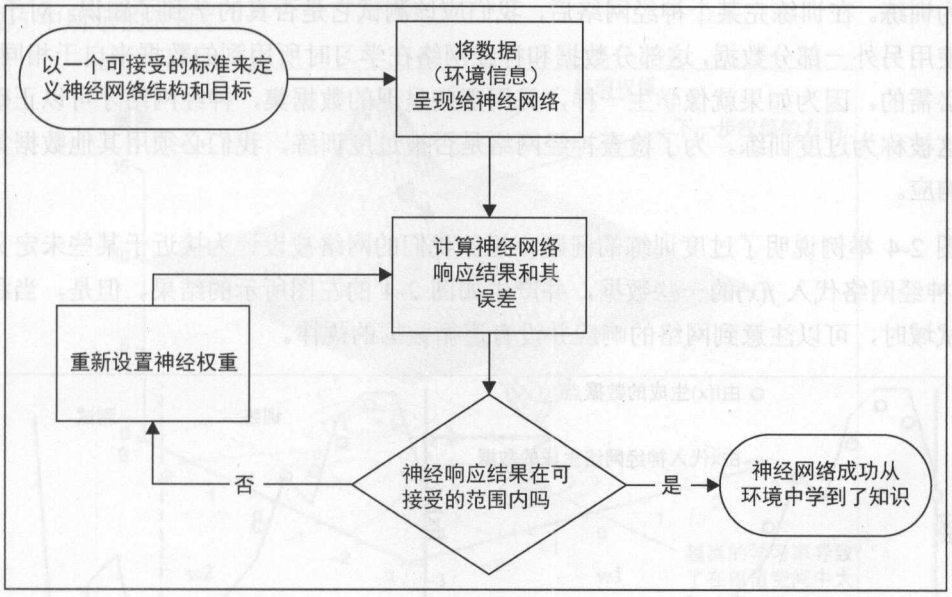


图 2-3

就像我们希望编写的程序那样，我们应该定义目标。所以，在这里，我们关注的是学习知识的神经网络。应该将知识（或者环境信息）呈现给人工神经网络，并检查它的响应结果，虽然结果通常是毫无意义的。网络响应的结果接下来将与期望结果相比较，并且结果将被代入到代价函数 C 中。代价函数将决定如何更新权值 W。接下来学习算法将计算 ΔW 项，这意味着会增加权值的变化。权值将按照以下等式进行更新。

$$W(k+1)=W(k)+\Delta W$$

k 代表第 k 次迭代，$W(k)$代表第 k 次迭代的神经权值，同理，$k+1$ 代表下一次迭代。

当学习过程开始，神经网络必须提供越来越接近于期望值的结果，直到最后，结果达到了期望的标准，学习过程就被认为结束了。

2.3.1 学习的两个阶段——训练和测试

那么，我们现在或许会问神经网络是否已经从数据中学习到了知识，但是我们如何证明它已经从数据中有效地学习到了知识？答案就像要求学生参加的考试一样，我们需要检验训练过后网络响应的结果。但是等等！你认为这就像一个老师总是在考试中设置那些出现在课堂中的那些问题？使用已知的案例来评估某人的学习效果没有什么意义，或者一个怀疑的老师会得出学生可能只是死记硬背而非理解这些内容。

好了，让我们来解释一下这一部分。这里正在讨论的是测试，已经涉及到的学习过程被称为训练。在训练完某个神经网络后，我们应该测试它是否真的学到了知识。对于测试，必须使用另外一部分数据，这部分数据和神经网络在学习时所用到的数据来自于相同环境。这是必需的，因为如果就像学生一样，只用那些呈现的数据集，神经网络才可以正确地响应，这被称为过度训练。为了检查神经网络是否被过度训练，我们必须用其他数据集来验证其响应。

图 2-4 举例说明了过度训练的问题。想象我们的网络被设计为接近于某些未定义函数 $f(x)$。神经网络代入 $f(x)$的一些数据，并产生如图 2-4 的左图所示的结果。但是，当我们扩大测试域时，可以注意到网络的响应并没有遵循数据的规律。

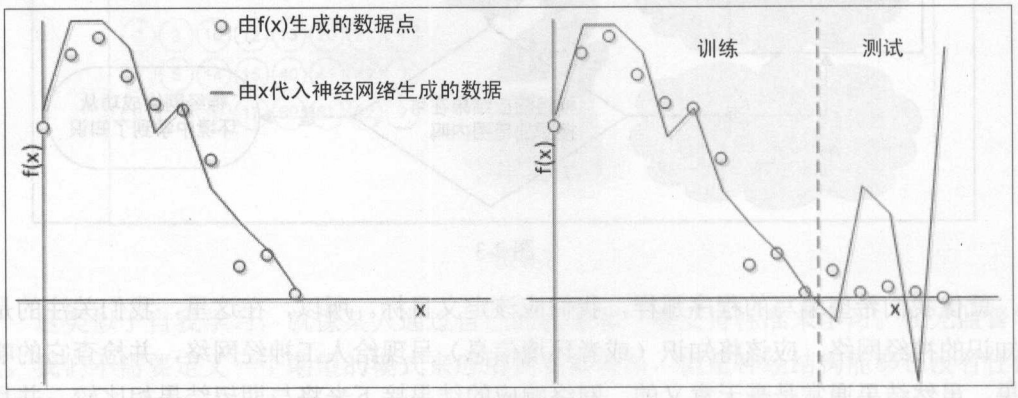


图 2-4

在这个例子中，我们看到了神经网络学习整个环境（函数 $f(x)$）的失败案例。可能是由以下几个原因导致此次失败：

- 神经网络没有从环境中得到足够的信息；
- 来源于环境的数据是非确定性的；
- 训练和测试数据集定义不明确；

- 神经网络从训练数据中学习了很多但忽视了测试数据。

在本书中，我们将介绍这个过程以防止这种情况的发生以及训练中可能出现的其他问题。

2.3.2 细节——学习参数

学习过程是可控的。一个重要的参数就是学习率，经常表示为希腊字母 η 。这个参数规定了神经网络权值在权值的多维空间中变化的强度。设想一个简单的神经网络，有两个输入单元和一个神经元以及一个输出单元。因此我们得到了两个权值 w1 和 w2。现在假设我们想要训练这个神经网络，那么是否可以评估每对权值的误差？假设我们应用的是一个如图 2-5 中所示的表面。

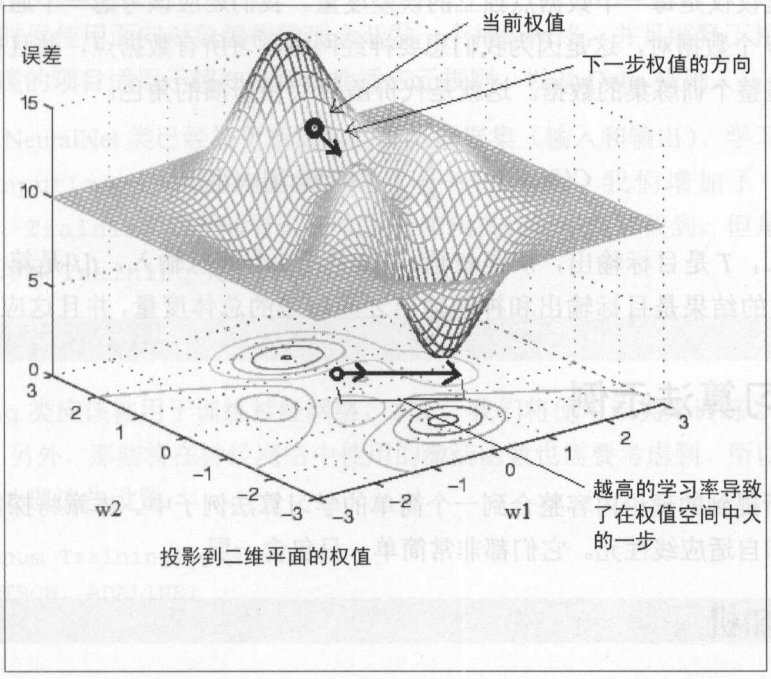


图 2-5

学习率负责调节权值在表面移动的距离。该参数可能加速学习过程但也可能导致一组比前一个更差的权值。

另一个重要的参数是停止条件。通常，当误差达到广义平均误差时，训练停止。但是在有些情况下，神经网络无法学习，权值几乎没有变化。此时，停止的条件是最大迭代次数。

2.3.3 误差度量和代价函数

误差度量和代价函数对于监督学习中训练过程的成败极其重要。假设我们有为神经网络准备的 N 条记录的数据集，包含了 X 和 T 变量，其中 X 为输出独立的值，T 为依赖于 X 的目标值。将神经网络认为是一个数学函数 $ANN()$，当它得到 X 值时会生成 Y 作为输出。

$$Y=ANN(x)$$

对于每一个给定到 ANN 的 x 值，都会生成一个 y 值，与 t 值相比较得到一个误差 e。

$$E=y-t$$

但是，这仅仅是每一个数据点独立的误差度量。我们还应该考虑一个通用的度量，它覆盖了所有 N 个数据对，这是因为我们想要神经网络学习所有数据点，并且相同的权值必须能生成覆盖整个训练集的数据。这就是代价函数 C 所扮演的角色。

$$C(X,T,W)=\frac{1}{N}\sum_{i=1}^{n=N}[ANN(x[i])-t[i]]^2$$

X 是输入，T 是目标输出，W 是权值，$x[i]$是第 i 个即时输入，$t[i]$是第 i 个即时目标输出。该函数的结果是目标输出和神经输出之间误差的总体度量，并且这应该被最小化。

2.4 学习算法示例

将前面所提到的理论内容整合到一个简单的学习算法例子中。本章将探索两个神经架构：感知机和自适应线性元。它们都非常简单，只包含一层。

2.4.1 感知机

感知机学习只需要考虑目标和输出的误差以及学习率。权值更新规律如下：

$$\Delta w_i=\eta(t[k]-y[k])x_i[k]$$

w_i 连接了第 i 个输入和神经元，$t[k]$是第 k 次样本数据的目标输出，$y[k]$是第 k 次样本的神经网络输出，$x_i[k]$是第 k 次样本的第 i 个输入，η 是学习率。可以看到规则刻意地简化并且不用考虑那些非线性激活函数的感知机，它只是简单地往误差的反方向逼近并理想地认为这能使网络接近于目标。

2.4.2 Delta 规则

有一个基于梯度下降的更优化的算法被开发出来以考虑非线性及其导数。这个算法增加到感知器规则中的是激活函数 $g(h)$的导数，h 是在应用到激活函数之前的所有神经元输入的加权综合，所以权值的更新规律如下：

$$\Delta w_i = \eta(t[k] - y[k])x_i[k]g'(h_i[k])$$

2.5 神经网络学习过程的编码

现在，是时候使用面向对象编程的理念开发一个神经网络，并且解释下相关理论了。前面章节中出现的项目适用于感知机和线性适应元规则，以及 Delta 规则。

前面章节 NeuralNet 类已经被更新到包含训练数据集（输入和输出）、学习参数和激活函数设置。`InputLayer` 类也被更新并包含了一个方法。我们增加了 `Adaline`、`Perceptron`、`Training` 类的设计，每个类的实现在代码中能够找到。但是，现在，让我们将神经学习和 `Training` 类的 Java 实现联系起来。

2.5.1 参数学习实现

`Training` 类应该被用于训练神经网络。本章，我们将使用该类来训练 `Perceptron` 和 `Adaline`。另外，那些将在神经网络中使用的激活函数也需要考虑到。所以让我们定义两个枚举集来处理这些设置：

```
public enum TrainingTypesENUM {
  PERCEPTRON, ADALINE;
}

public enum ActivationFncENUM {
  STEP, LINEAR, SIGLOG, HYPERTAN;
}
```

除了这些参数，我们还需要定义停止条件、误差、MSE 误差和迭代次数，如下面代码所示：

```
private int epochs;
private double error;
private double mse;
```

学习率已经在 NeuralNet 类中定义过了，这里将会使用。

最后，需要一个方法来更新某个指定神经元的权值，所以让我们先来看看 CalcNewWeight 方法：

```
private double calcNewWeight(TrainingTypesENUM trainType,
    double inputWeightOld, NeuralNet n, double error,
    double trainSample, double netValue) {
  switch (trainType) {
  case PERCEPTRON:
    return inputWeightOld + n.getLearningRate() * error * trainSample;
  case ADALINE:
    return inputWeightOld + n.getLearningRate() * error * trainSample
      * derivativeActivationFnc(n.getActivationFnc(), netValue);
  default:
    throw new IllegalArgumentException(trainType
        + " does not exist in TrainingTypesENUM");
  }
}
```

这个方法中有一个 switch 语句，可以根据训练类型（Adaline 或者 Perceptron）。我们还看见 inputWeightOld（老的权值）、n（神经网络训练次数）、error（目标和神经输出的误差）、trainsample（给权值的输入）、netValue（激活函数处理之前的加权总和）等参数。学习率通过调用 NeuralNet 类的 getLearningRate()方法进行获取。

一个有趣的细节是激活函数的导数被 Adaline 训练类型所调用，这就是 Delta 规则。所有的激活函数都在 Training 类中被实现为方法，它们相应的导数也在 Training 类中实现。derivativeActivationFnc 方法帮助调用作为参数传入的激活函数相应的导数。

2.5.2 学习过程

在 Training 类中实现的两个特殊方法：一个是用来训练神经网络，另一个是用来训练某些层的神经元。尽管这在本章不是必需的，但为将来的例子和更新做好代码准备总是好的。让我们来快速浏览下这个训练方法：

```
public NeuralNet train(NeuralNet n) {

    ArrayList<Double> inputWeightIn = new ArrayList<Double>();

    int rows = n.getTrainSet().length;
    int cols = n.getTrainSet()[0].length;

    while (this.getEpochs() < n.getMaxEpochs()) {

      double estimatedOutput = 0.0;
      double realOutput = 0.0;

      for (int i = 0; i < rows; i++) {

        double netValue = 0.0;

        for (int j = 0; j < cols; j++) {
          inputWeightIn = n.getInputLayer().getListOfNeurons().get(j)
              .getListOfWeightIn();
          double inputWeight = inputWeightIn.get(0);
          netValue = netValue + inputWeight * n.getTrainSet()[i][j];
        }

        estimatedOutput = this.activationFnc(n.getActivationFnc(),
            netValue);
        realOutput = n.getRealOutputSet()[i];

        this.setError(realOutput - estimatedOutput);

        if (Math.abs(this.getError()) > n.getTargetError()) {
          // fix weights
          InputLayer inputLayer = new InputLayer();
          inputLayer.setListOfNeurons(this.teachNeuronsOfLayer(cols,
              i, n, netValue));
          n.setInputLayer(inputLayer);
        }
      }

      this.setMse(Math.pow(realOutput - estimatedOutput, 2.0));
      n.getListOfMSE().add(this.getMse());
```

```
        this.setEpochs(this.getEpochs() + 1);
    }

    n.setTrainingError(this.getError());

    return n;
}
```

这个方法接收一个神经网络作为入参，然后返回另一个神经网络，它的权值已经被训练过。接下来看 while 子句，当迭代次数不等于 Training 类中设置的最大迭代次数时，将继续循环。在这个循环中，有一个 for 子句，迭代了神经网络所有的训练样本，并在当前迭代中开始计算对应输入的神经输出的过程。

当得到了真实的神经网络输出，就将其与估计的输出比较下，并计算误差。检查误差，如果误差高于最小误差，则通过调用 teachNeuronsOfLayer 开始更新，代码如下：

```
inputLayer.setListOfNeurons(this.teachNeuronsOfLayer(cols,
            i, n, netValue));
```

这个方法的实现在本章所附的代码中可以找到。

接下来，这个过程会不断重复，直到神经网络接收了所有的样本数据，当达到最大迭代次数，循环结束。

2.5.3 类定义

表 2-1 展示了本章设计的所有类的字段和方法。

表 2-1

类名：Training	
注意：该类为抽象类且不能被实例化	
属性	
private int epochs	用来存储训练周期的次数，被称为 epoch
private double error	存储预期输出和实际输出的实数
private double mse	存储均方误差的实数（MSE）
枚举	
注意：枚举有助于控制不同类型	

续表

枚举	

public enum TrainingTypesENUM {
PERCEPTRON, ADALINE;
}
	存储项目支持的训练类型（Perceptron 和 Adaline）

public enum ActivationFncENUM {
STEP, LINEAR, SIGLOG,
HYPERTAN;
}
	存储项目支持的激活函数类型（阶跃、线性、sigmoid logistics、双曲正切）

方法	
public NeuralNet train(NeuralNet n)	训练神经网络
	参数：NeuralNet 对象(未训练的神经网络)
	返回值：NeuralNet 对象 (训练过的神经网络)
public ArrayList<Neuron> teachNeuronsOfLayer(int numberOfInputNeurons, int line, NeuralNet n, double netValue)	使某层的神经元计算并改变其权值
	参数：输入神经元的数量、样本数量、NeuralNet 对象、神经网络净输出
	返回值：ArrayList 集合，元素类型为 Neuron
private double calcNewWeight(TrainingTypesENUM trainType, double inputWeightOld, NeuralNet n, double error, double trainSample, double netValue)	计算某个神经元的新权值
	参数：训练类型的枚举值、老的输入权值、NeuralNet 对象、误差值、训练样本值、净输出值
	返回值：代表新权值的实数
public double activationFnc (ActivationFncENUM fnc, double value)	决定要使用的激活函数，并调用其计算方法
	参数：激活函数枚举值、实数值
	返回值：激活函数计算结果值

续表

方法	
public double derivativeActivationFnc (ActivationFncENUM fnc, double value)	决定选择哪个激活函数并调用计算其导数的方法
	参数：激活函数枚举值、实数值
	返回值：激活函数的导数的计算结果
private double fncStep (doublev)	计算阶跃函数
	参数：实数值
	返回值：实数值
private double fncLinear (double v)	计算线性函数
	参数：实数值
	返回值：实数值
private double fncSigLog (double v)	计算 sigmoid logistics 函数
	参数：实数值
	返回值：实数值
private double fncHyperTan (double v)	计算双曲正切函数
	参数：实数值
	返回值：实数值
private double derivativeFncLinear (double v)	计算线性函数的导数
	参数：实数值
	返回值：实数值
private double derivativeFncSigLog (double v)	计算 sigmoid logistics 函数的导数
	参数：实数值
	返回值：实数值
private double derivativeFncHyperTan (doublev)	计算双曲正切函数的导数
	参数：实数值
	返回值：实数值

续表

方法	
public void printTrainedNetResult (NeuralNet trainedNet)	输出训练过的神经网络并显示其结果
	对象：NeuralNet 对象
	返回值：None
public int getEpochs()	返回训练的迭代次数
public void setEpochs (int epochs)	设置训练的迭代次数
public double getError()	返回训练误差（估计值与真实值比较）
public void setError (double error)	设置训练误差
public double getMse()	返回 MSE
public void setMse (double mse)	设置 MSE
Java 实现类 Training.java 文件	
类名：Perceptron	
注意：该类从 Training 类继承其方法和属性	
属性	
None	
方法	
public NeuralNet train(NeuralNet n)	使用感知机算法训练神经网络
	参数：NeuralNet 对象（未训练的神经网络）
	返回值：NeuralNet 对象（通过感知机算法训练后的 NeuralNet 对象）
Java 实现类 Perceptron.java 文件	
类名：Adaline	
注意：该类从 Training 类继承其方法和属性	
属性	
None	

续表

方法	
public NeuralNet train(NeuralNet n)	使用 adaline 算法训练神经网络
	参数：NeuralNet 对象（未训练的神经网络）
	返回：NeuralNet 对象（通过 adaline 算法训练后的 NeuralNet 对象）
Java 实现类：Adaline.java 文件	
类名：InputLayer	
注意：该类在前面的版本已经出现过，现更新如下	
属性	
None	
方法	
public void setNumberOfNeuronsInLayer(int numberOfNeuronsInLayer)	设置输出层的神经元数量，由于偏置的存在，它是一个一个增加的
Java 实现类：InputLayer.java 文件	
类名：NeuralNet	
注意：该类在前面的版本已经出现过，现更新如下	
属性	
private double[][] trainSet	存储输入数据的训练集矩阵
private double[] realOutputSet	存储输出数据的训练集向量
private int maxEpochs	存储神经网络训练的最大迭代次数
private double learningRate	存储学习率的实数
private double targetError	存储目标误差的实数
private double trainingError	存储训练误差的实数
private TrainingTypesENUM trainType	训练神经网络的训练类型枚举值

续表

属性	
private ActivationFncENUM activationFnc	用于训练的激活函数枚举值
private ArrayList<Double> listOfMSE = new ArrayList<Double>()	存储每次迭代的 MSE 的实数 ArrayList
方法	
public NeuralNet trainNet (NeuralNet n)	训练神经网络
	参数：NeuralNet 对象（未训练的神经网络）
	返回值：NeuralNet 对象（训练的神经网络）
public void printTrainedNetResult (NeuralNet n)	输出神经网络并显示其结果
	参数：NeuralNet 对象
	返回值：None
public double[][] getTrainSet()	返回输入数据的训练集矩阵
public void setTrainSet(double[][] trainSet)	设置输入数据的训练集矩阵
public double[] getRealOutputSet()	返回输出数据的训练集向量
public void setRealOutputSet(double[] realOutputSet)	设置输出数据的训练集向量
public int getMaxEpochs()	返回神经网络将要训练的最大迭代次数
public void setMaxEpochs(int maxEpochs)	设置神经网络将要训练的最大迭代次数
public double getTargetError()	返回目标误差
public void setTargetError(double targetError)	设置目标误差

续表

方法	
`public double getLearningRate()`	返回训练中使用的学习率
`public void setLearningRate(double learningRate)`	设置训练中使用的学习率
`public double getTrainingError()`	返回目标误差
`public void setTrainingError(double trainingError)`	设置目标误差
`public ActivationFncENUM getActivationFnc()`	返回训练中使用的激活函数枚举值
`public void setActivationFnc( ActivationFncENUM activationFnc)`	设置训练中使用的激活函数枚举值
`public TrainingTypesENUM getTrainType()`	返回训练中使用的训练类型枚举值
`public void setTrainType( TrainingTypesENUM trainType)`	设置训练中使用的训练类型枚举值
`public ArrayList<Double> getListOfMSE()`	返回存储每次迭代的 MES 误差的实数列表
`public void setListOfMSE( ArrayList<Double> listOfMSE)`	设置存储每次迭代的 MES 误差的实数列表
Java 实现类：NeuralNet.java 文件	

更新的类图如图 2-6 所示，此处省略前一章节已经解释的属性和方法。此外，还省略了新属性（setter 和 getter）的配置方法。

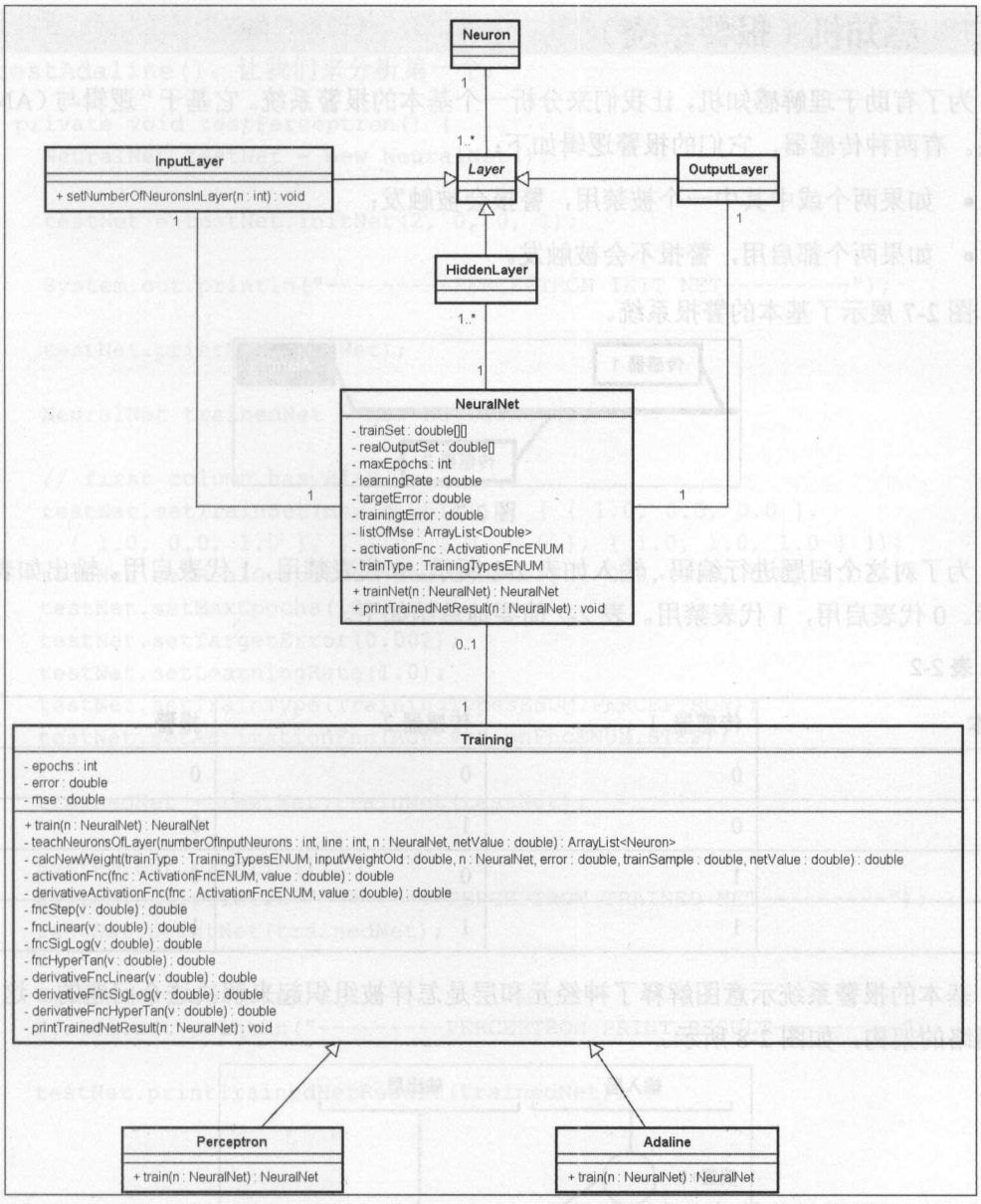


图 2-6

2.6 两个实例

现在，让我们来看看这些简单的神经网络架构的应用实例。

2.6.1 感知机（报警系统）

为了有助于理解感知机，让我们来分析一个基本的报警系统。它基于“逻辑与（AND）”关系。有两种传感器，它们的报警逻辑如下：

- 如果两个或中其中一个被禁用，警报会被触发；
- 如果两个都启用，警报不会被触发。

图 2-7 展示了基本的警报系统。

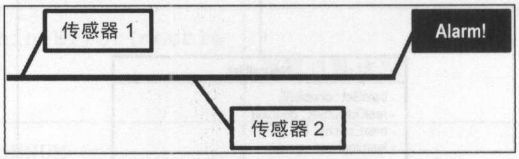


图 2-7

为了对这个问题进行编码，输入如表 2-2 所示。0 代表禁用，1 代表启用。输出如表 2-2 所示。0 代表启用，1 代表禁用。表 2-2 简要地总结如下。

表 2-2

样本	传感器 1	传感器 2	报警
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1

基本的报警系统示意图解释了神经元和层是怎样被组织起来解决这个问题的。这是神经网络的架构，如图 2-8 所示。

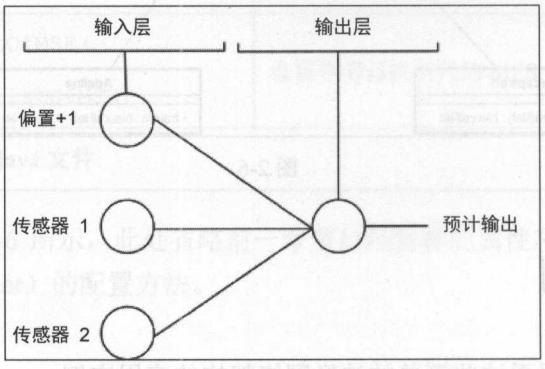


图 2-8

现在，让我们使用先前引用的类。在测试类中已经创建了两个方法：testPerceptron()和testAdaline()。让我们来分析第一个：

```
private void testPerceptron() {
  NeuralNet testNet = new NeuralNet();

  testNet = testNet.initNet(2, 0, 0, 1);

  System.out.println("---------PERCEPTRON INIT NET---------");

  testNet.printNet(testNet);

  NeuralNet trainedNet = new NeuralNet();

  // first column has BIAS
  testNet.setTrainSet(new double[][] { { 1.0, 0.0, 0.0 },
    { 1.0, 0.0, 1.0 }, { 1.0, 1.0, 0.0 }, { 1.0, 1.0, 1.0 } });
  testNet.setRealOutputSet(new double[] { 0.0, 0.0, 0.0, 1.0 });
  testNet.setMaxEpochs(10);
  testNet.setTargetError(0.002);
  testNet.setLearningRate(1.0);
  testNet.setTrainType(TrainingTypesENUM.PERCEPTRON);
  testNet.setActivationFnc(ActivationFncENUM.STEP);

  trainedNet = testNet.trainNet(testNet);

  System.out.println();
  System.out.println("---------PERCEPTRON TRAINED NET---------");
  testNet.printNet(trainedNet);

  System.out.println();
  System.out.println("---------PERCEPTRON PRINT RESULT---------");

  testNet.printTrainedNetResult(trainedNet);

}
```

首先，创建一个 NeuralNet 对象。之后，用这个对象来初始化神经网络，这个神经网络输入层有两个神经元，隐层有一个神经元，输出层有一个神经元。接下来，一条消息和未训练的神经网络会显示在屏幕上。在那之后，testNet 对象设置训练输入数据集（第一列为偏置）、训练输出数据集、最大迭代次数、目标误差、学习率、学习类型（感知机）和激活函数（阶跃函数）。然后，调用 trainNet 方法训练该神经网络。最后，打印感知机训练网络结果。这些结果被显示在图 2-9 所示的屏幕截图上。

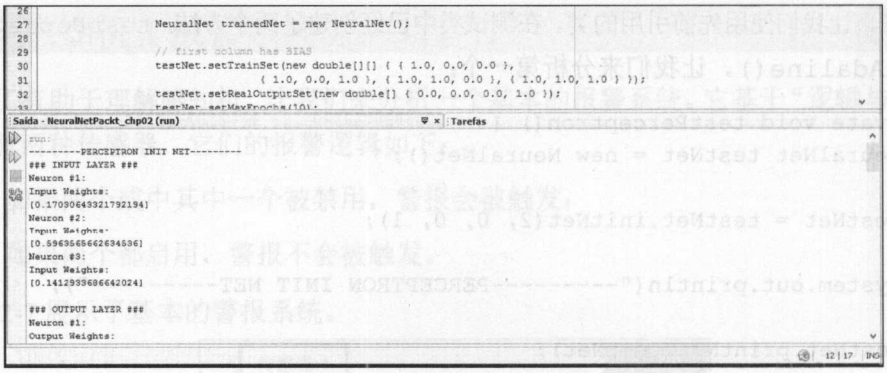


图 2-9

```
---------PERCEPTRON INIT NET---------
### INPUT LAYER ###
Neuron #1:
Input Weights:
[0.179227246819473]
Neuron #2:
Input Weights:
[0.927776315380873]
Neuron #3:
Input Weights:
[0.7639255282026901]

### OUTPUT LAYER ###
Neuron #1:
Output Weights:
[0.7352957201253741]

---------PERCEPTRON TRAINED NET---------
### INPUT LAYER ###
Neuron #1:
Input Weights:
[-2.820772753180527]
Neuron #2:
Input Weights:
[1.9277763153808731]
Neuron #3:
Input Weights:
[1.76392552820269]

### OUTPUT LAYER ###
Neuron #1:
Output Weights:
[0.7352957201253741]

---------PERCEPTRON PRINT RESULT---------
```

```
1.0 0.0 0.0 NET OUTPUT: 0.0 REAL OUTPUT: 0.0 ERROR: 0.0
1.0 0.0 1.0 NET OUTPUT: 0.0 REAL OUTPUT: 0.0 ERROR: 0.0
1.0 1.0 0.0 NET OUTPUT: 0.0 REAL OUTPUT: 0.0 ERROR: 0.0
1.0 1.0 1.0 NET OUTPUT: 1.0 REAL OUTPUT: 1.0 ERROR: 0.0
```

根据结果，就可以检查权值是否改变，并总结出神经网络学习如何对何时启用和禁用警报进行分类。提醒：获得的知识蕴含在这些权值里：[-2.820772753180527]、[1.9277763153808731]、[1.76392552820269]。另外，用伪随机数来初始化神经元，每次代码运行，结果可能变化。

2.6.2 ADALINE（交通预测）

为了展示自适应线性元算法，假设城市的某一小部分有一条大街，3 条街道通向这条大街。在大街上，交通情况拥挤，发生了不少交通事故。假设政府交通部门决定开发一个预测和报警的新系统。这个系统旨在预测交通拥堵，提醒驾驶员，并采取必要的措施来减少已发生的损失，如图 2-10 所示。

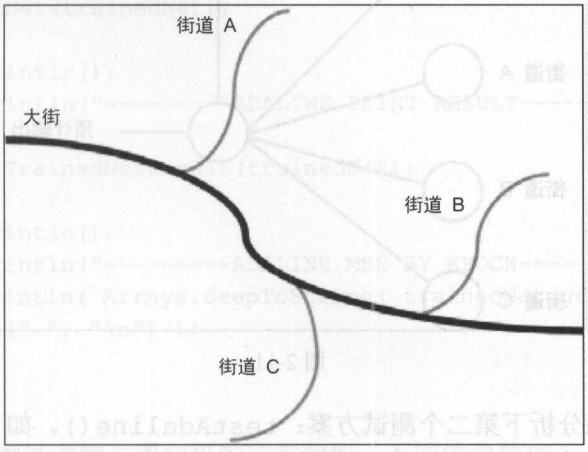


图 2-10

为了开发这个系统，我们收集了每条街道和大街近一周的信息：每条路每分钟的车流量，如表 2-3 所示。

表 2-3

样本	街道 A(每分钟车流量)	街道 B(每分钟车流量)	街道 C(每分钟车流量)	大街(每分钟车流量)
1	0.98	0.94	0.95	0.80
2	0.60	0.60	0.85	0.59
3	0.35	0.15	0.15	0.23

续表

样本	街道 A(每分钟车流量)	街道 B(每分钟车流量)	街道 C(每分钟车流量)	大街(每分钟车流量)
4	0.25	0.30	0.98	0.45
5	0.75	0.85	0.91	0.74
6	0.43	0.57	0.87	0.63
7	0.05	0.06	0.01	0.10

接下来，解决这个问题的神经网络架构的设计方案如图 2-11 所示。

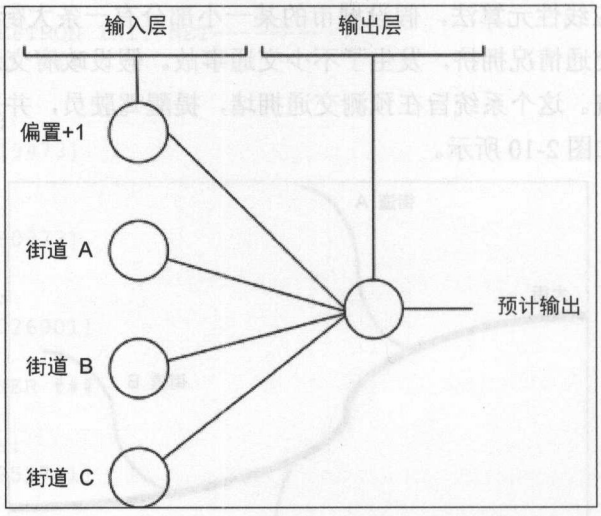


图 2-11

接下来，我们来分析下第二个测试方案：testAdaline()。如下：

```
private void testAdaline() {

  NeuralNet testNet = new NeuralNet();

  testNet = testNet.initNet(3, 0, 0, 1);

  System.out.println("---------ADALINE INIT NET---------");

  testNet.printNet(testNet);

  NeuralNet trainedNet = new NeuralNet();

  // first column has BIAS
```

```
        testNet.setTrainSet(new double[][] { { 1.0, 0.98, 0.94, 0.95 },
            { 1.0, 0.60, 0.60, 0.85 }, { 1.0, 0.35, 0.15, 0.15 },
            { 1.0, 0.25, 0.30, 0.98 }, { 1.0, 0.75, 0.85, 0.91 },
            { 1.0, 0.43, 0.57, 0.87 }, { 1.0, 0.05, 0.06, 0.01 } });
        testNet.setRealOutputSet(new double[] { 0.80, 0.59, 0.23, 0.45, 0.74, 0.63,
0.10 });
        testNet.setMaxEpochs(10);
        testNet.setTargetError(0.0001);
        testNet.setLearningRate(0.5);
        testNet.setTrainType(TrainingTypesENUM.ADALINE);
        testNet.setActivationFnc(ActivationFncENUM.LINEAR);

        trainedNet = new NeuralNet();
        trainedNet = testNet.trainNet(testNet);

        System.out.println();
        System.out.println("---------ADALINE TRAINED NET---------");

        testNet.printNet(trainedNet);

        System.out.println();
        System.out.println("---------ADALINE PRINT RESULT---------");

        testNet.printTrainedNetResult(trainedNet);

        System.out.println();
        System.out.println("---------ADALINE MSE BY EPOCH---------");
        System.out.println( Arrays.deepToString( trainedNet.getListOfMSE().
toArray() ).replace(" ", "\n") );
    }
```

自适应线性元的测试逻辑与感知机的非常相似。不同的参数如下：输入层的 3 个神经元、测试数据集、输出数据集、训练类型集合如自适应线性元、激活函数激活如线性激活函数。最后，输出自适应线性元训练结果和自适应线性元均方误差，如图 2-12 所示。

```
The complete results are displayed via following code:
---------ADALINE INIT NET---------
### INPUT LAYER ###
Neuron #1:
Input Weights:
[0.39748670958336774]
Neuron #2:
Input Weights:
[0.0018141925587737973]
```

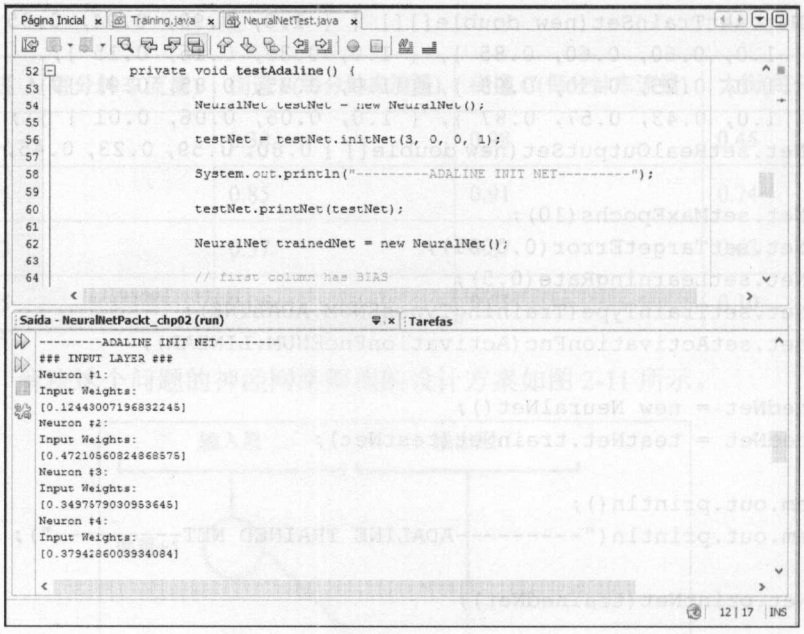


图 2-12

```
Neuron #3:
Input Weights:
[0.3705005221910509]
Neuron #4:
Input Weights:
[0.20624007274978795]

### OUTPUT LAYER ###
Neuron #1:
Output Weights:
[0.16125863508860827]

---------ADALINE TRAINED NET---------
### INPUT LAYER ###
Neuron #1:
Input Weights:
[0.08239521813153253]
Neuron #2:
Input Weights:
[0.08060471820877586]
Neuron #3:
Input Weights:
[0.4793193652720801]
```

```
Neuron #4:
Input Weights:
[0.259894055603035]

### OUTPUT LAYER ###
Neuron #1:
Output Weights:
[0.16125863508860827]

---------ADALINE PRINT RESULT---------
1.0    0.98  0.94 0.95 NET OUTPUT: 0.85884 REAL OUTPUT: 0.8
ERROR: 0.05884739815477136
1.0    0.6   0.6 0.85 NET OUTPUT:  0.63925 REAL OUTPUT: 0.59
ERROR: 0.04925961548262592
1.0    0.35  0.15 0.15 NET OUTPUT: 0.22148 REAL OUTPUT: 0.23 ERROR:
-0.008511117364128656
1.0    0.25  0.3  0.98 NET OUTPUT: 0.50103 REAL OUTPUT: 0.45
ERROR: 0.05103838175632486
1.0    0.75  0.85 0.91 NET OUTPUT: 0.78677 REAL OUTPUT: 0.74
ERROR: 0.046773807868144446
1.0    0.43  0.57 0.87 NET OUTPUT: 0.61637 REAL OUTPUT: 0.63
ERROR: -0.013624886458967755
1.0    0.05  0.06 0.01 NET OUTPUT: 0.11778 REAL OUTPUT: 0.1 ERROR:
0.017783556514326462

---------ADALINE MSE BY EPOCH---------
[0.04647154331286084,
0.018478851884998992,
0.008340477769290564,
0.004405551259806042,
0.0027480838150394362,
0.0019914963464723553,
0.0016222114177244264,
0.00143318844904685,
0.0013337070214879325,
0.001280852868781586]
```

根据上述的结果，可以得出结论：神经网络在一个特定区域学习了预测交通拥堵。这可以通过改变权值和均方误差（MSE）列表来证明。图 2-13 是使用均方误差（MSE）数据进行的数据可视化，很容易发现当迭代次数升高，均方误差（MSE）则下降。

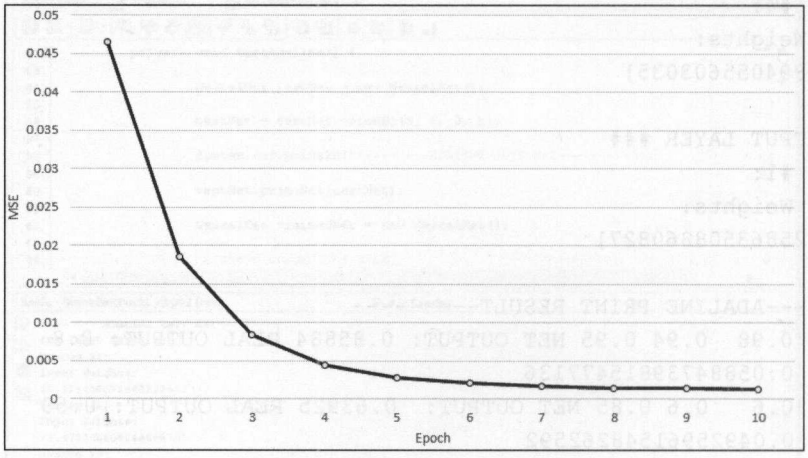


图 2-13

2.7 小结

本章向读者介绍了神经网络的整个学习过程。我们展现了学习的基础，它的灵感来自于人类自身的学习。为了用实践来说明这个过程，我们用 Java 实现了两个学习算法并将其应用到两个实例中。通过这些，读者可以对神经网络如何学习有一个虽然基础但是有用的理解，甚至可以系统地描述这个学习过程。这是阅读下一章的基础，下一章将会展示更多复杂的例子。

第 3 章
运用感知机

本章将会探索最流行也是最基础的神经网络架构之一——感知机。本章也会介绍它们的扩展版本（也被称为多层感知机），以及它们的特性、学习算法和参数。同时，读者将会学习如何用 Java 来实现和使用它们以解决一些基本的问题。

- 感知机
  - 应用和局限
- 多层感知机
  - 分类
  - 回归
- 反向传播算法
- Java 实现
- 实际问题

3.1 学习感知机神经网络

感知机是最简单的神经网络。1957 年由 Frank Rosenblatt 提出，它仅有一层神经元，接收一系列输入并产生一系列输出。这是最早获得关注的神经网络形式之一，特别是它的简洁性。一个简单的神经元的结构如图 3-1 所示。

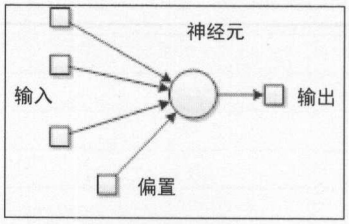


图 3-1

3.1.1 感知机的应用和局限性

然而，科学家没多久就得出结论，感知机神经网络由于其结构简单只能处理简单的分类任务，通常在面对更加复杂的数据集时一筹莫展。让我们来回顾下第 2 章的第一个例子（AND 指的是“逻辑与”），来更好地理解这个问题。

3.1.2 线性分离

这个例子包含了一个与（AND）函数，它接收两个输入 $x1$ 和 $x2$。这个函数可以用一个二维图表来描绘，如图 3-2 所示。

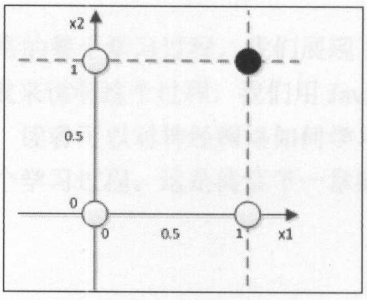


图 3-2

现在，我们来看看神经网络如何使用感知机规则在训练过程中不断进化的。如表 3-1 所示，一个权值对：$w1$ 和 $w2$，初始化为 0.5，偏置为 0.5。假设学习率 η=0.2。

表 3-1

Epoch	*x*1	*x*2	*w*1	*w*2	*b*	*y*	*t*	*E*	Δ*w*1	Δ*w*2	Δ*b*
1	0	0	0.5	0.5	0.5	0.5	0	−0.5	0	0	−0.1
1	0	1	0.5	0.5	0.4	0.9	0	−0.9	0	−0.18	−0.18
1	1	0	0.5	0.32	0.22	0.72	0	−0.72	−0.144	0	−0.144
1	1	1	0.356	0.32	0.076	0.752	1	0.248	0.0496	0.0496	0.0496

续表

Epoch	x1	x2	w1	w2	b	y	t	E	Δw1	Δw2	Δb
2	0	0	0.406	0.370	0.126	0.126	0	−0.126	0.000	0.000	−0.025
2	0	1	0.406	0.370	0.100	0.470	0	−0.470	0.000	−0.094	−0.094
2	1	0	0.406	0.276	0.006	0.412	0	−0.412	−0.082	0.000	−0.082
2	1	1	0.323	0.276	−0.076	0.523	1	0.477	0.095	0.095	0.095
…	…										
89	0	0	0.625	0.562	−0.312	−0.312	0	0.312	0	0	0.062
89	0	1	0.625	0.562	−0.25	0.313	0	−0.313	0	−0.063	−0.063
89	1	0	0.625	0.500	−0.312	0.313	0	−0.313	−0.063	0	−0.063
89	1	1	0.562	0.500	−0.375	0.687	1	0.313	0.063	0.063	0.063

在第 89 次迭代后，我们发现神经网络产生的值趋近于期望输出。这是由于在这个例子中，输出值是二值化（0 或 1）的，我们可以假设神经网络产生的任何小于 0.5 的值都认为是 0，任何大于 0.5 的值都认为是 1。所以，我们可以用由学习算法得到的最终参数 $w1 = 0.562$, $w2 = 0.5$ 和 $b = -0.375$ 确定函数 $Y = x1w1 + x2w2 + b = 0.5$ 来定义这个线性边界，如图 3-3 所示。

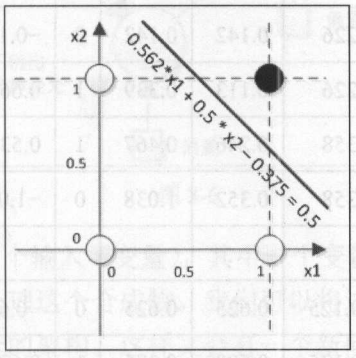


图 3-3

这个边界是对由神经网络得到的所有分类的定义。你可以看见这个边界是线性的，这是由于函数是线性的。因此，感知机网络特别适合那些线性可分的分类模式。

3.1.3 经典 XOR（异或）例子

让我们来分析下 XOR（异或）的例子，它的图表如图 3-4 所示。

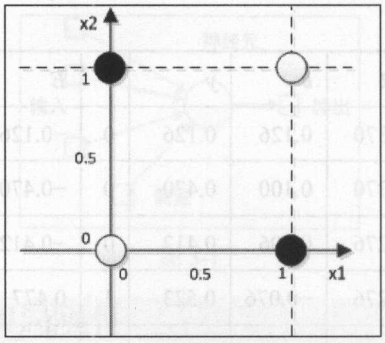


图 3-4

我们可以看见在两个维度中，是不可能画一条线就将两个模式分开的。如果试图训练一个单层感知机来学习这个函数会发生什么呢？那我们来试试，可以从表 3-2 看看发生了什么。

表 3-2

Epoch	*x*1	*x*2	*w*1	*w*2	*b*	*y*	*t*	*E*	Δ*w*1	Δ*w*2	Δ*b*
1	0	0	0.5	0.5	0.5	0.5	0	−0.5	0	0	−0.1
1	0	1	0.5	0.5	0.4	0.9	1	0.1	0	0.02	0.02
1	1	0	0.5	0.52	0.42	0.92	1	0.08	0.016	0	0.016
1	1	1	0.516	0.52	0.436	1.472	0	−1.472	−0.294	−0.294	−0.294
2	0	0	0.222	0.226	0.142	0.142	0	−0.142	0.000	0.000	−0.028
2	0	1	0.222	0.226	0.113	0.339	1	0.661	0.000	0.132	0.132
2	1	0	0.222	0.358	0.246	0.467	1	0.533	0.107	0.000	0.107
2	1	1	0.328	0.358	0.352	1.038	0	−1.038	−0.208	−0.208	−0.208
…	…										
127	0	0	−0.250	−0.125	0.625	0.625	0	−0.625	0.000	0.000	−0.125
127	0	1	−0.250	−0.125	0.500	0.375	1	0.625	0.000	0.125	0.125
127	1	0	−0.250	0.000	0.625	0.375	1	0.625	0.125	0.000	0.125
127	1	1	−0.125	0.000	0.750	0.625	0	−0.625	−0.125	−0.125	−0.125

感知机不能找到使误差低于 0.625 的权值对。这可以从数学的角度来解释，我们从这个图表已经发现这个函数在两个维度上是线性不可分的。所以，如果再增加另一个维度会怎么样？让我们看看前面的 3 个维度的 XOR（异或）图表，如图 3-5 所示。

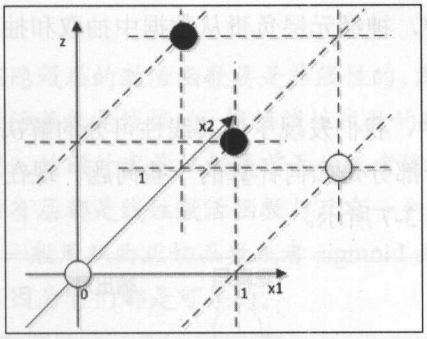


图 3-5

在 3 个维度中，可以画出一个平面，这个平面可以将模式分开，前提是这个额外的维度可以对输入数据进行恰当的变换。好了，现在又有一个额外的问题：由于只有两个输入变量，我们又如何得到这个额外的维度。一个显而易见的权宜之计是增加第三个变量作为两个原生变量的衍生。通过第三个变量（衍生维度），我们的神经网络可能变成如下形状，如图 3-6 所示。

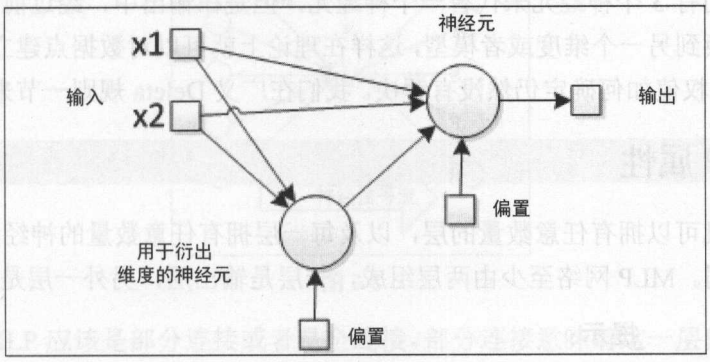


图 3-6

好了，现在，感知机有 3 个输入（变量），其中一个变量由另外两个变量合成。这也引起了另外一个问题：该如何处理这个合成物。我们可以将这部分视为一个神经元，如此一来，会赋予神经网络一个嵌套的架构。这样又会有一个新问题：当误差体现在输出神经元时，如何训练新的神经元的权值。

3.2 流行的多层感知机（MLP）

正如我们所见，在这个简单的例子中，模式并不是线性可分的，这导致了我们对感知机架构的使用有了越来越多的问题。在第 1 章中，我们知道了神经网络也是按照层来进行

组织的。在人工神经网络中，神经元层负责从数据中抽取和抽象信息，将其转换到另一个维度中或者模型里。

在XOR（异或）例子中，我们发现了使其线性可分的解决方案是增加第三个部分。但是，这会留下关于这第三个部分是如何计算的一些问题。现在，让我们考虑用一个两层感知机来作为代替方案，如图3-7所示。

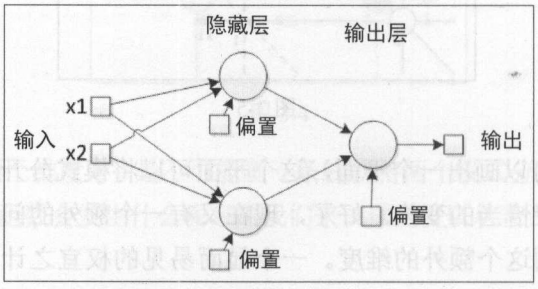


图3-7

现在，我们有3个神经元来代替一个神经元，但是在输出中，经过前面层传递过来的信息已经被转换到另一个维度或者模型，这样在理论上就可以对数据点建立一个线性边界。但是，第一层的权值如何确定仍然没有解决。我们在广义Deleta规则一节来处理这个问题。

3.2.1 MLP属性

多层感知机可以拥有任意数量的层，以及每一层拥有任意数量的神经元，每一层的激活函数可以不同。MLP网络至少由两层组成，一层是输出层，另外一层是隐藏层。

提示：

有一些文献认为输入层是收集数据的节点。所以，在这些案例中，MLP被认为至少拥有3层。在本书中，我们认为输入层是一个没有权值的特殊层，而那些有效层，也就是可以被训练的部分，我们认为是隐藏层和输出层。

隐藏层之所以被称为隐藏层，是因为它确实是对外部世界“隐藏”了它的输出。隐藏层可以以任意数量串联而成，因而形成了深度神经网络。但是神经网络层越多，训练和运行就越慢，并且根据数据理论基础，一到两个隐藏层的神经网络可以训练得像那些有很多隐藏层的深度神经网络。

提示：

建议在隐藏层的激活函数要是非线性的，尤其是当输出层的激活函数是线性的。根据线性代数的理论，倘若通过层引入的额外变量仅仅是前面的或者输入的线性组合，所有层都是线性激活函数与只有一个输出层等效。通常，一般用双曲正切函数或者 sigmoid 函数作为激活函数，因为它们都是可导的。

3.2.2 MLP 权值

在一个 MLP 前馈网络中，某个神经元 i 从前一层的神经元 j 接收数据，并向前传播它的输出到下一层的神经元 k，如图 3-8 所示。

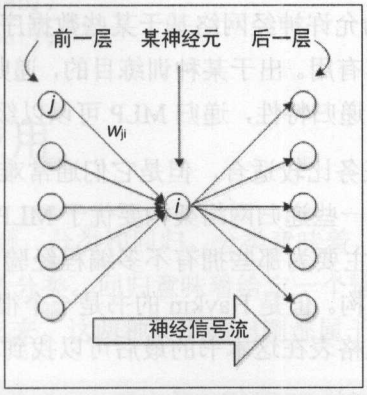


图 3-8

理论上的 MLP 应该是部分连接或者是全连接。部分连接意味着这一层的神经元并不是都与下一层的每一个神经元相连接，而全连接意味着这一层的神经元与下一层的每一个神经元相连。图 3-9 展示了部分连接层和全连接层。

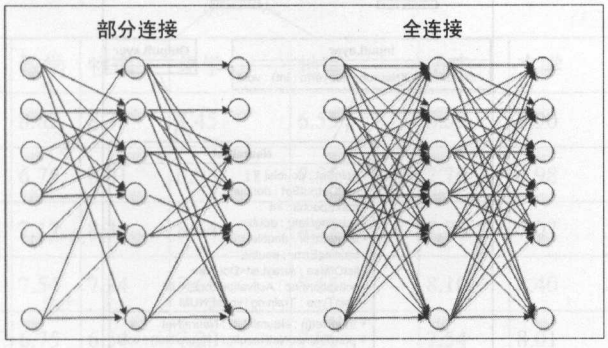


图 3-9

为了简化数学理论，我们只考虑全连接 MLP，这可以用下面的数学等式描述：

$$y_o = f_o\left(\sum_{i=1}^{n_{hl}} w_i f_i\left(\sum_{j=1}^{n_{hl-1}} w_i f_i\left(\sum_{k=1}^{n_{hl-2}} w_{jk} f_k(\cdots)+b_j\right)+b_i\right)+b_o\right)$$

y_o 是神经网络的输出（如果我们有多个输出，我们可以用 Y 来代替 y_O 来表示一个向量），f_o 是输出的激活函数，l 是隐藏层的数量，n_{hi} 是隐藏层 i 的神经元数量，w_i 是连接最后的隐藏层的第 i 个神经元到输出层的权值，f_i 是神经元 i 的激活函数，b_i 是神经元 i 的偏置。它可以随着神经元层数的增加而变大。在最后的求和操作中，输入参数就是 x_i。

3.2.3 递归 MLP

神经网络既可以是前馈型又是反馈（递归）型。所以，一些神经元或者层将信号传递给前一层是可能的。这种行为允许神经网络基于某些数据序列保持状态，这种特性在处理时间序列或者手写识别时尤其有用。出于某种训练目的，递归 MLP 可以只在输出层有反向连接。为了赋予它更加完全的递归特性，递归 MLP 可以以级联的方式连接多个递归 MLP。

尽管递归网络对于某些任务比较适合，但是它们通常难以训练，最后，计算机可能会在运行时内存溢出。另外，有一些递归网络架构要优于 MLP，例如 Elman、Hopfeld、回声状态和双向 RNN。由于本书主要为那些拥有不多编程经验的读者而关注那些最简单的应用，因此我们不会深究这些架构。但是 Haykin 的书是一个很好的参考，对于那些感兴趣的读者，关于递归神经网络的规格表在这本书的最后可以找到。

3.2.4 MLP 在 OOP 范式中的结构

将这些概念融入到 OOP 的概念，我们可以审视下目前为止已经设计的类，得到图 3-10 这张类图。

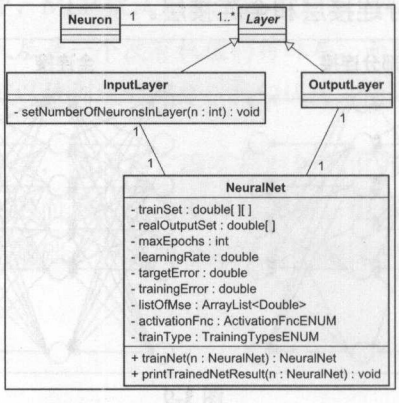


图 3-10

可以看出神经网络的结构是分层。神经网络是由层构成，层是由神经元构成。在 MLP 架构，有 3 种层类型：输入、隐藏和输出。所以，假设用 Java，我们会定义一个包含 3 个神经元的输入层、包含一个神经元的输出层和包含 5 个神经元的隐藏层。代码如下：

```
NeuralNet n = NeuralNet();
InputLayer input = new InputLayer();
input.setNumberOfNeuronsInLayer(3);
HiddenLayer hidden = new HiddenLayer();
hidden.setNumberOfNeuronsInLayer(5);
OutputLayer output = new OutputLayer();
output.setNumberOfNeuronsInLayer(1);
////…
n.setInputLayer(input);
n.setHiddenLayer(hidden);
n.setOutputLayer(output);
```

3.3 有趣的 MLP 应用

MLP 适合处理两大类问题：分类和回归。分类意味着，给定一个数据集，其中的每一条记录都应该被打上标签或者分类。回归意味着给定一个输入和输出的集合，必须找到一个函数能将输入和输出映射起来。这两种类型的问题都属于监督学习的范畴。

3.3.1 使用 MLP 进行分类

给定一个类和数据的集合，根据历史数据集包含的记录和它们相应的类别，某人想对其进行分类。表 3-3 展示了样例数据集，科目的平均等级范围是 0～10。

表 3-3

学号	英语	数学	物理	化学	地理	历史	文学	生物	职业
89543	7.82	8.82	8.35	7.45	6.55	6.39	5.90	7.03	电气工程师
93201	8.33	6.75	8.01	6.98	7.95	7.76	6.98	6.84	市场专员
95481	7.76	7.17	8.39	8.64	8.22	7.86	7.07	9.06	医生
94105	8.25	7.54	7.34	7.65	8.65	8.10	8.40	7.44	律师
96305	8.05	6.75	6.54	7.20	7.96	7.54	8.01	7.86	校长

续表

学号	英语	数学	物理	化学	地理	历史	文学	生物	职业
92904	6.95	8.85	9.10	7.54	7.50	6.65	5.86	6.76	程序员
…	…	…							

一个例子是基于学业成绩来预测其职业。我们参考一个关于已经工作的毕业生数据集。我们编制了一个数据集，包含了每个人上学时每个科目的平均等级和他/她现在的职业。注意，输出是职业的名称，但是神经网络并不能直接给出。我们需要为每个已知职业新增一列（输出）来代替。如果选择了某个职业，相应职业的那一列的值为1；反之为0。以下展示了这个矩阵的一部分：

各科成绩　　职业

$$\begin{bmatrix} \overbrace{7.82 & \cdots & 7.03}^{\text{各科成绩}} & \overbrace{1 & \cdots & 0}^{\text{职业}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 5.66 & \cdots & 6.22 & 0 & \cdots & 1 \end{bmatrix}$$

现在，我们想基于某个人的成绩来预测某人将来会选择哪个职业。最后，我们搭建的神经网络的输入层包含了科目数目的神经元数量，输出层包含了已知职业数目的神经元，隐藏层包含了任意多个隐藏神经元。这个问题的神经网络模式如图3-11所示。

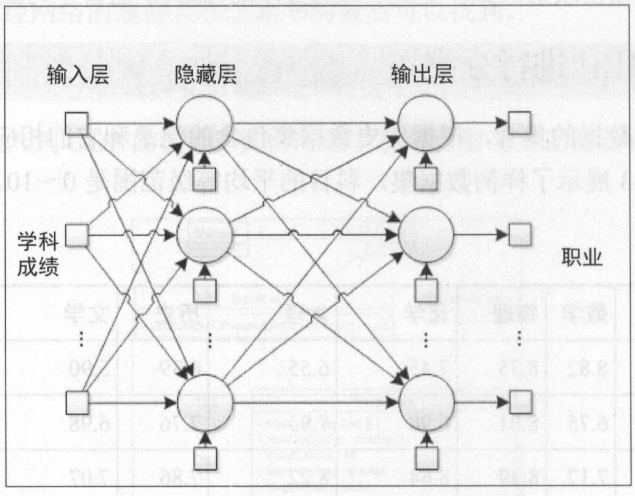


图3-11

对于这个分类问题，对于每个数据点通常只有一个类。所以，在输出层，神经元被激发产生的值非0即1，可以用激活函数将输出值映射到0～1的区间要好些。但是，我们必

须考虑一种情况，那就是不止一个神经元被激发，将两个类赋予一条记录。有一些机制可以防止这种情况，例如 softmax 函数或者赢家通吃算法。这些机制会在第 6 章中详细介绍。

在经过训练后，神经网络已经学习了在给定人员的给定成绩下，最有可能的职业。

3.3.2 用 MLP 进行回归

回归包括找到映射一系列输入和一系列输出的函数问题。表 3-4 展示了绑定到 n 个相关输出的 k 条记录的 m 维独立输入 X 的数据集。

表 3-4

m 维独立输入 X 的数据集				n 维输出数据集			
X1	X2	…	XM	T1	T2	…	TN
x1[0]	x2[0]	…	xm[0]	t1[0]	t2[0]	…	tn[0]
x1[1]	x2[1]	…	xm[1]	t1[1]	t2[1]	…	tn[1]
…	…	…	…	…	…	…	…
x1[k]	x2[k]	…	xm[k]	t1[k]	t2[k]	…	tn[k]

之前的表可以被编辑成矩阵格式

$$[X \quad T]$$

这里：

$$X_{k,m} = \begin{bmatrix} x_1[0] & x_2[0] & \cdots & x_m[0] \\ x_1[1] & x_2[1] & \cdots & x_m[1] \\ \vdots & \vdots & \ddots & \vdots \\ x_1[k] & x_2[k] & \cdots & x_m[k] \end{bmatrix}$$

$$T_{k,n} = \begin{bmatrix} t_1[0] & t_2[0] & \cdots & t_n[0] \\ t_1[1] & t_2[1] & \cdots & t_n[1] \\ \vdots & \vdots & \ddots & \vdots \\ t_1[k] & t_2[k] & \cdots & t_n[k] \end{bmatrix}$$

和分类不太一样，输出值是数值化的而不是标签或者类别。还是会有一个包含我们希望神经网络学习的一些行为记录的历史数据库。一个例子是预测两个城市之间的巴士票价。在这个例子中，我们收集了一系列城市和当前城市之间的巴士票价的信息。我们用城市之间的距离和（或者）行驶时间来构建城市特征作为输入，巴士票价作为输出。图 3-12 展示

了城市之间的道路网，城市用字母表示。

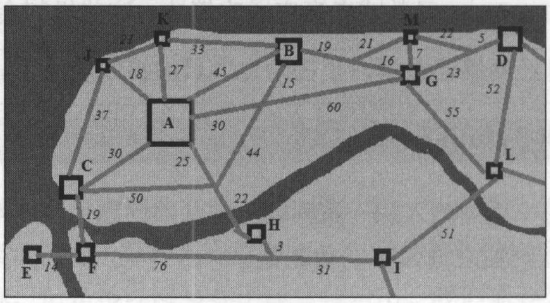


图 3-12

表 3-5 展示了从图 3-12 中提到的城市中得到的信息，并将其结构化为适合神经网络的格式。

表 3-5

起点城市特征			终点城市特征			路线特征			票价
人口	GDP	路线	人口	GDP	路线	距离	时间	停靠次数	
500000	4.5	6	45000	1.5	5	90	1.5	0	15
120000	2.6	4	500000	4.5	6	30	0.8	0	10
30000	0.8	3	65000	3.0	3	103	1.6	1	20
35000	1.4	3	45000	1.5	5	7	0.4	0	5
…									
120000	2.6	4	12000	0.3	3	37	0.6	0	7

结构化这个数据集后，我们定义了一个 MLP 网络，输入层的神经元个数为一个精确值（由于有两个城市为城市特征数乘以 2）加上道路特征数，输出层的神经元个数为 1，隐藏层的神经元个数为任意值。表 3-5 呈现的例子中，神经元一共有 9 个。由于输出神经元是数值化的，因此不需要限制输出层，故而选用线性函数作为输出层的激活函数会更合适些。神经网络会对两个城市之间的路线给出一个估计的价格，哪怕这条路线目前还没有被任何巴士传输公司运营。

3.4 MLP 的学习过程

多层感知机网络基于 Delta 规则的基础进行学习，这也受了梯度下降优化法的启发。梯度方法广泛应用在寻找函数极值上。一个基于搜索方法的梯度进化如图 3-13 所示。

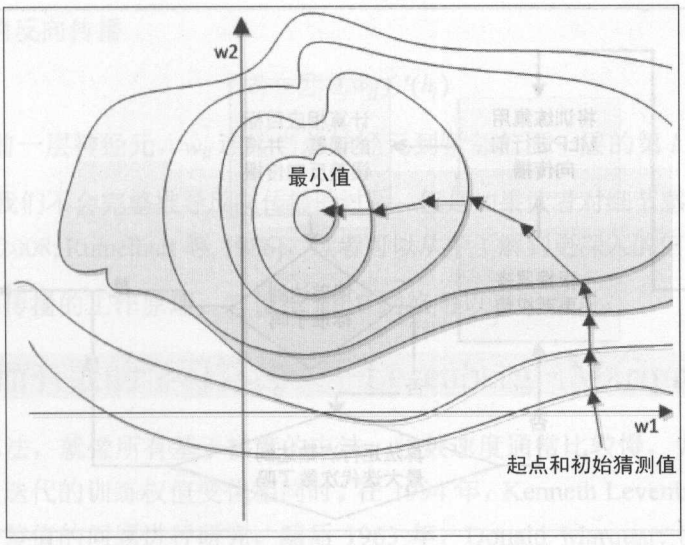


图 3-13

这个方法应用在根据一定的条件，函数输出趋向于越来越大或者越来越小。这个概念是对 Delta 规则的探索。

$$\Delta w_i = \eta(t[k] - y[k])x_i[k]g'(h_i[k])$$

Delta 规则想要最小化的函数是神经网络输出和目标输出的误差，参数则是神经网络的权值。这与感知机规则相比是增强学习算法，因为它考虑了激活函数的导数 $g'(h)$，它用数学上的术语来说就是指向了函数下降得最快的方向。

3.4.1 简单但很强大的学习算法——反向传播

尽管 Delta 原则对于那些只有输入层和输出层的神经网络来说很有效，但由于隐藏层的存在，纯粹的 Delta 规则却不能应用于 MLP 网络。为了克服这一点，在 1980 年，Rummelhart 等人提出了一个新的算法，也是受梯度方法所启发，被称为反向传播。

这个算法对于 MLP 来说确实是 Delta 规则的泛化。用额外的层来抽象更多从环境中获得的数据，这个优点促进了训练算法的发展，使其能正确地调整隐藏层的权值。基于梯度方法的基础，输出的误差会传播到前面的层，从而使得使用与 Delta 规则相同的方程来更新权值成为可能。算法按照图 3-14 所示的流程图运行。

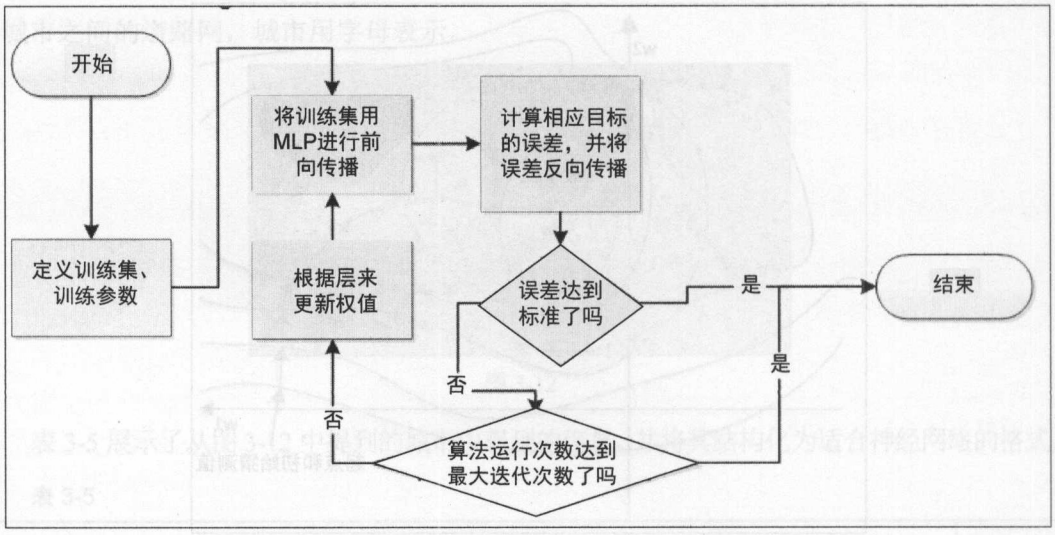


图 3-14

第二步是反向传播自身。它做的是基于 Deleta 规则根据梯度找到权值的变化。

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial h_i}\frac{\partial h_i}{\partial w_{ji}} = (t-y)f'(h_i)x_j$$

E 是误差，w_{ij} 是神经元 i 和 j 之间的权值，o_i 是第 i 个神经元的输出，h_i 是传到激活函数之前的神经元输入的加权总和。$o_i = f(h_i)$，f 是激活函数。

更新隐藏层的工作有点复杂，因为我们将误差视为要更新的权值和输出之间的所有神经元的一个函数。为了简化这个过程，我们先计算敏感性或者反向传播误差：

$$\delta_i = \frac{\partial E}{\partial v_i}\frac{\partial o_i}{\partial h_i}$$

接下来，权值更新如下：

$$\Delta w_{ji} = -\eta\frac{\partial E}{\partial w_{ji}} = -\eta\delta_i x_j$$

对输出层和隐藏层计算反向传播误差变化如下：

- 输出层的反向传播

$$\delta_i = (o_i - t_i)f'(h_i)$$

  - o_i 是第 i 个输出，t_i 是期望的第 i 个输出，$f'(h_i)$ 是输出激活函数的导数，h_i 是第 i 个输入神经元的加权总和。

- 隐藏层的反向传播

$$\delta_i = \sum_l \delta_l w_{il} f'(h_i)$$

  - l 是前一层神经元，w_{il} 连接当前神经元到紧邻前面一层的第 l 个神经元的权值。

为了简化，我们不会完整推导反向传播的过程。但是如果读者对细节感兴趣，我们推荐参考文献 [Haykin, 2008; Rumelhart 等, 1986]，读者可以从中了解到更深入的信息。

这就是反向传播的工作原理，它使得 MLP 网络可以进行学习。

3.4.2 复杂而有效的学习算法——Levenberg - Marquardt

反向传播算法，就像所有基于梯度的方法，收敛速度通常比较慢，尤其当它在之字型路线以及每两次迭代的训练权值变化相同时。在 1994 年，Kenneth Levenberg 将这个缺点作为类似曲线拟合差值的问题进行研究，随后 1963 年，Donald Marquart 也开始研究，他开发了一个基于 Gauss - Newton 算法和梯度下降算法来找到系数的方法，这就是这个算法名字的来源。

该算法处理的一些优化项超出了本书的范围，但在本书的参考章节，有一些很好的资源，读者可以从中了解到更多关于这些概念的信息，所以我们用一种更简单的方式来表示这种方式。假设我们一个输入 x 和输出 t 的集合：

$$\begin{bmatrix} x_1[0] & x_2[0] & \cdots & x_m[0] & t_1[0] & t_2[0] & \cdots & t_n[0] \\ x_1[1] & x_2[1] & \cdots & x_m[1] & t_1[1] & t_2[1] & \cdots & t_n[1] \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1[k] & x_2[k] & \cdots & x_m[k] & t_1[k] & t_2[k] & \cdots & t_n[k] \end{bmatrix}$$

我们已经看到，神经网络具有将输入映射到输出的属性，就像非线性函数 f 和系数 W（权值和偏置）：

$$Y=f(X,W)$$

非线性函数将会生成不同于输出 T 的值，这是因为在方程中我们标记了变量 Y。Levenberg-Marquardt 算法基于 Jacobian 矩阵进行研究，对于每一条数据，这个矩阵包含了每一个权值和偏置的所有偏导数。所以，Jacobian 矩阵格式如下：

$$J = \begin{bmatrix} \frac{\partial f(X[1],W)}{W_1} & \cdots & \frac{\partial f(X[1],W)}{W_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(X[k],W)}{W_1} & \cdots & \frac{\partial f(X[k],W)}{W_p} \end{bmatrix}$$

k 是所有数据点的数量，p 是所有权值和偏置的数量。在 Jacobian 矩阵，一个数据点所有权值和偏置偏导权被连续存储在一行。Jacobian 矩阵的元素可以从梯度计算得到：

$$\frac{\partial E}{\partial w_{ji}} = (t-y)\frac{\partial f(x_i,W)}{\partial w_{ji}} \Rightarrow \frac{\partial f(x_i,W)}{\partial w_{ji}} = \frac{\partial E}{\partial w_{ji}}(t-y)^{-1}$$

误差 E 的每个权值的偏导数可以由反向传播算法算出，所以这个算法同样会运行反向传播算法。

在每一个优化问题里，都想最小化总误差：

$$E(W) = \sum [Y_i - f(X_i,W)]^2$$

W（在神经网络这个例子中代表权值和偏置）是要优化的变量。优化算法通过增加 ΔW 来更新 W。根据一些代数理论，我们可以扩展上面那个方程，如下：

$$E(W+\Delta W) = \sum [Y_i - f(X_i,W) - J_i \Delta W]^2$$

通过转变成向量和符号，我们可以得到：

$$E(W+\Delta W) = \| Y - f(X,W) - J\Delta W \|^2$$

最后，通过设置误差 E 为 0，经过一些变换后，我们得到了 Levenberg-Marquardt 等式：

$$\Delta W = (J^T J + \lambda I)^{-1} J^T (Y - f(X,W))$$

这就是权值更新的规则。正如我们所看到的，它包含了矩阵操作，例如转置和逆。希腊字母 λ 是阻尼因素，相当于学习率。

3.5 MLP 实现

现在，让我们来实现曾讨论过的所有理论。这里使用定义了 ANN 结构的 `Neuralnet` 类、`Layer` 类、`Neuron` 类等。现在，增加 `HiddenLayer` 和 `OutputLayer` 类（这两个类继承了 `Layer` 类）的函数来实现多层神经网络。

我们也实现了在本章出现的两个学习算法——反向传播和 Levenberg-Marquardt。在 `Training` 这个类中，我们增加了两个 `Training` 枚举类型：`BACKPROPAGATION` 和 `LEVENBERG_ MARQUARDT`。

为了实现 Levenberg - Marquardt 算法，我们引入了 edu.packt.neuralnet.util 包和两个类——Matrix 类、IdentityMatrix 类。这些类实现了应用到 Levenberg - Marquardt 算法的矩阵操作。但是，我们不会深入研究这两个类，只是使用基本的矩阵操作。

表 3-6 展示了本章使用的类的相关属性和方法。

表 3-6

类名：Training	
注意：该类为抽象类且不能被实例化	
枚举	
注意：枚举有助于控制不同的类型	

public enum TrainingTypesENUM {
PERCEPTRON, ADALINE,
BACKPROPAGATION;
}
	用来存储项目支持的训练类型（添加了反向传播）

类名：**Backpropagation**	
注意：该类从 Training 类继承属性和方法	
属性	
None	
方法	
public NeuralNet train (NeuralNetn)	使用反向传播算法训练神经网络。该方法重写了 Training 类的 train 方法
	Parameters：NeuralNet 对象 (未训练的神经网络)
	返回值：NeuralNet 对象(经过反向传播算法训练的神经网络)
private NeuralNet forward (NeuralNet n, int row)	从第一层开始传播信号到隐藏层再到输出层
	参数：NeuralNet 对象, 训练集的行号
	返回值：NeuralNet 对象

续表

方法	
private NeuralNet backpropagation (NeuralNet n, int row)	从输出层反向传播信号到隐藏层再到第一层。在该方法中，权值会被调整
	参数：NeuralNet 对象，训练集的行号
	返回值：NeuralNet 对象
Java 实现类：Backpropagation.java 文件	
类名：LevenbergMarquardt	
注意：该类从 Backpropagation 类继承属性和方法	
属性	
private double dampingFactor	阻尼系数，类似于学习率
private Matrix jacobian	在 Levenberg-Marquardt 算法中使用的 Jacobian 矩阵
方法	
public NeuralNet train (NeuralNetn)	使用 Levenberg–Marquardt 算法训练神经网络，该方法重写了 Backpropagation 类的 train 方法。
	参数：NeuralNet 对象(未训练的神经网络)
	返回值：NeuralNet 对象(通过反向传播算法训练的神经网络)
public void buildJacobianMatrix (NeuralNet n, int row)	通过训练数据集的相应行，为神经网络的每个权值和偏置计算梯度并保存到 Jacobian 矩阵相应的行里。
	参数：NeuralNet 对象(未训练的神经网络)和 row (第 i 个数据点)
	返回值：无
类名：NeuralNet	
注意：该类在前面的版本已经出现过，现更新如下	
属性	
private double[][] realMatrixOutputSet	存储训练集的输出数据（矩阵格式）

续表

属性	
`private double errorMean`	存储两个或更多神经元之间的平均误差的实数
`private ActivationFncENUM activationFncOutputLayer`	在网络的输出层会使用的激活函数的枚举值。
方法	
注意：这些属性的 get 和 set 方法也已经被创建	
Java 实现类：NeuralNet.java 文件	

类图的改动如图 3-15 所示，属性和方法在前面的章节已经解释过了，并且省略了它们的配置方法（getter 和 setter）。

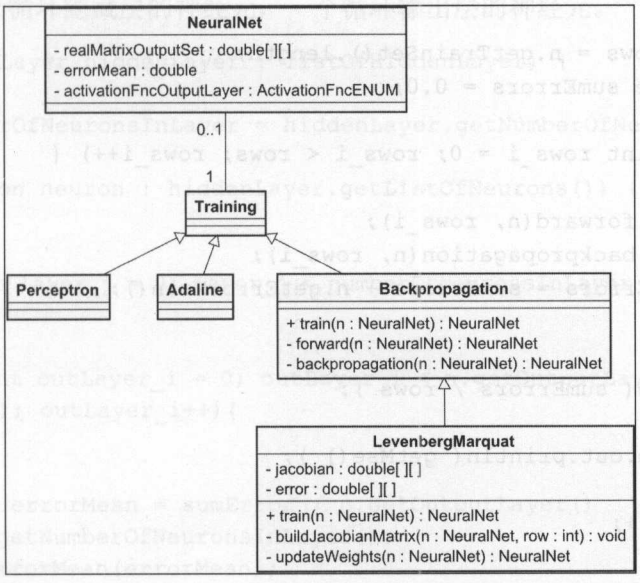


图 3-15

3.5.1 实战反向传播算法

我们已经从流程图看到了反向传播算法一共有两个步骤：

- 传播神经信号；
- 反向传播误差。

所以，backpropagation 类为每个步骤准备了两个特殊方法：forward()方法和backpropagation()方法。backpropagation 类的 train()方法将会调用这两个方法。

3.5.2 探索代码

让我们来分析 forward()、backpropagation()、train()方法。Train()方法会调用 forward()方法和 backpropagation()方法。

```
public NeuralNet train(NeuralNet n) {

    int epoch = 0;
    setMse(1.0);

    while(getMse() > n.getTargetError()) {

      if ( epoch >= n.getMaxEpochs() ) break;

      int rows = n.getTrainSet().length;
      double sumErrors = 0.0;

      for (int rows_i = 0; rows_i < rows; rows_i++) {

        n = forward(n, rows_i);
        n = backpropagation(n, rows_i);
        sumErrors = sumErrors + n.getErrorMean();
      }

      setMse( sumErrors / rows );

      System.out.println( getMse() );

      epoch++;
    }

    System.out.println("Number of epochs: "+epoch);

    return n;
}
```

首先，这段代码得到训练参数，设置 MSE（代表均方误差），MSE 是训练的停止条件。第一个循环停止条件是 MSE 小于目标误差。另外，在循环中，还要一个跳出条件，即当前的迭代次数达到最大迭代次数。

第二个循环会遍历训练集的每一个数据点，对每个数据点重复训练过程，并首次调用 forward()函数和 backpropagation()函数，在本节前面已经详细介绍过这两个函数了。误差会被汇总。当遍历完训练集的所有数据点后，该方法设置当前的 MSE，将其打印到屏幕，并将迭代数加 1。

现在，让我们来分析 forward()函数和 backpropagation()函数。因为它们代码很长，所以我们将会探索其最重要的部分。

forward()函数执行从输入层到输出层的神经计算。为了简单起见，该实现只处理一个隐藏层和一个输出层的情况，与多个隐藏层神经网络相比，这种简单的架构被证明工作得相当好。该函数接收一个 NeuralNet 对象和数据集中要被传播的数据点的行号作为参数。

```
private NeuralNet forward(NeuralNet n, int row)
```

它初始化了一些参数，例如误差总和、估计与实际输出。大体上，一个大循环包含了两个小循环，一个循环隐藏层的神经元，一个循环输出层的神经元。

```
for (HiddenLayer hiddenLayer : listOfHiddenLayer) {

  int numberOfNeuronsInLayer = hiddenLayer.getNumberOfNeuronsInLayer();

  for (Neuron neuron : hiddenLayer.getListOfNeurons()) {


    for (int layer_j = 0; layer_j < numberOfNeuronsInLayer - 1; layer_j++) {
     }

     for (int outLayer_i = 0; outLayer_i < n.getOutputLayer().getNumberOf
NeuronsInLayer(); outLayer_i++){
     }

     double errorMean = sumError / n.getOutputLayer()
          .getNumberOfNeuronsInLayer();
     n.setErrorMean(errorMean);

     n.getListOfHiddenLayer().get(hiddenLayer_i)
          .setListOfNeurons(hiddenLayer.getListOfNeurons());
  }
}
```

在计算了隐藏层和输出层的输出后，这个函数最后会计算反向传播用的误差。隐藏层和输出层的计算细节在本章相应的源代码中。

反向传播函数也接收一个 NeuralNet 对象和数据集中要被传播的数据点的行号作为参数。

```
private NeuralNet backpropagation(NeuralNet n, int row)
```

为了更容易理解，该函数被分为6个部分。

1. 初始化训练参数并检索神经网络层（隐藏层和输出层）。
2. 计算输出层的敏感性。
3. 计算隐藏层的敏感性。
4. 更新输出层的权值。
5. 更新隐藏层的权值。
6. 更新神经网络中的层。

我们主要关注第2～5部分。输出层的敏感性计算非常简单，计算敏感性参数那一行向我们展示了Delta规则。

```
//sensibility output layer
for (Neuron neuron : outputLayer) {
  error = neuron.getError();
  netValue = neuron.getOutputValue();
  sensibility = derivativeActivationFnc(
      n.getActivationFncOutputLayer(), netValue ) * error;

  neuron.setSensibility(sensibility);
}
```

对于隐藏层来说，需要将输出层的权值和敏感性求和。在计算完敏感性后，局部变量tempSensibility处理求和。可以看到该参数在一个遍历了那层所有神经元的循环中被计算完成。

```
for (Neuron neuron : hiddenLayer) {

  sensibility = 0.0;

  if(neuron.getListOfWeightIn().size() > 0) { //exclude bias
    ArrayList<Double> listOfWeightsOut = new ArrayList<Double>();

    listOfWeightsOut = neuron.getListOfWeightOut();

    double tempSensibility = 0.0;

    int weight_i = 0;
```

```
        for (Double weight : listOfWeightsOut) {
          tempSensibility = tempSensibility + (weight *
              outputLayer.get(weight_i)
              .getSensibility());
          weight_i++;
        }

        sensibility = derivativeActivationFnc (
          n.getActivationFnc(), neuron.getOutputValue() ) *
          tempSensibility;

        neuron.setSensibility(sensibility);

      }
    }
```

更新输出层的权值和更新各自的敏感性一样简单。一个循环遍历连接到每个输出神经元的所有隐藏层神经元。局部变量 newWeight 负责为每个相应的权值介绍新的值。

```
    //fix weights (teach) [output layer to hidden layer]
    for (int outLayer_i = 0; outLayer_i < n.getOutputLayer().getNumberOfNeurons
InLayer(); outLayer_i++) {
      for (Neuron neuron : hiddenLayer) {
        double newWeight = neuron.getListOfWeightOut()
          .get( outLayer_i ) + ( n.getLearningRate() *
              outputLayer.get( outLayer_i )
              .getSensibility() *
              neuron.getOutputValue() );

        neuron.getListOfWeightOut().set(outLayer_i,
              newWeight);
      }
    }
```

对于隐藏层来说，使用的敏感性参数根据反向传播章节所示的方程，也需要一个内部循环来遍历所有输入神经元。

```
    //fix weights (teach) [hidden layer to input layer]
    for (Neuron neuron : hiddenLayer) {

      ArrayList<Double> hiddenLayerInputWeights = new ArrayList<Double>();
      hiddenLayerInputWeights = neuron.getListOfWeightIn();

      if(hiddenLayerInputWeights.size() > 0) { //exclude bias

        int hidden_i = 0;
```

```
        double newWeight = 0.0;
        for (int i = 0; i < n.getInputLayer().getNumberOfNeuronsInLayer(); i++) {

        newWeight = hiddenLayerInputWeights.get(hidden_i) +
            ( n.getLearningRate() *
              neuron.getSensibility() *
              n.getTrainSet()[row][i] );

        neuron.getListOfWeightIn().set(hidden_i, newWeight);

        hidden_i++;
        }
      }
    }
```

3.6 Levenberg - Marquardt 实现

Levenberg-Marquardt 使用许多反向传播算法的特性，这就是我们继承 backpropagation 类的原因。基本上，训练函数是相同的，除了下面的代码片段：

```
for (int rows_i = 0; rows_i < rows; rows_i++) {
  n = forward(n, rows_i);
  buildJacobianMatrix(n, rows_i);
  sumErrors = sumErrors + n.getErrorMean();
}
n=updateWeights(n);
```

这个循环遍历了所有数据集，对每条数据调用 buildJacobianMatrix 方法。这个方法调用了继承自 backpropagation 方法的原始版本来计算梯度。

从前面解释的 LMA 理论可以看出，Jacobian 矩阵的行依次包含了所有权值和偏置。所以，Jacobian 矩阵的权值的对应列的细节如表 3-7 所示。

表 3-7

层	权值或偏置	位置
隐藏层	第 i 个神经元的第 j 个权值	(i * (numberOfInputs)) + j
输出层	第 i 个神经元的偏置	((numberOfInputs) * (numberOfHiddenNeurons - 1)) + (i * (numberOfHiddenNeurons) + numberOfHiddenNeurons)

由于 buildJacobianMatrix 方法有点像 backpropagation 方法，我们要突出 Jacobian 行的构造。对于隐藏层的权值，下面的代码会被调用：

```
jacobian.setValue( row, ( i * ( numberOfInputs ) ) + j,
    ( neuron.getSensibility() *
    n.getTrainSet()[row][j] ) / nb.getErrorMean() );
```

我们看到在梯度中使用的隐藏层神经元的敏感性。现在，对于输出层，我们使用下面的代码：

```
jacobian.setValue( row,
    ( numberOfInputs + 1 ) * ( numberOfHiddenNeurons ) +
    ( i * ( numberOfHiddenNeurons + 1 ) ) + j,
    ( output.getSensibility() * neuron.getOutputValue() ) /
       n.getErrorMean() );
```

在这段代码中，神经元对象指的是在输出层之前的隐藏神经元。

反向传播算法和 Levenberg-Marquardt 算法还有一个区别是权值在一次迭代中被更新一次，而不是基于每个数据点。这是因为 Jacobian 是用整个数据集构建的。

在构建 Jacobian 矩阵后，我们能在训练方法后看到算法调用了 updateWeights 方法。用这种方法，可以解出 Levenberg - Marquardt 等式，接着，权值被增加到来自 delta 矩阵的对应贡献值上。

Levenberg-Marquardt 等式的解：

```
Matrix term1 = jacobian.transpose().multiply(jacobian)
    .add(new IdentityMatrix(jacobian.getNumberOfColumns())
    .multiply(damping));
Matrix term2 = jacobian.transpose().multiply(error);
Matrix delta = term1.inverse().multiply(term2);
```

更新隐藏层中的第 i 个神经元的第 j 个权值：

```
newWeight = hiddenLayerInputWeights.get( i ) +
    delta.getValue( ( i * ( numberOfInputs + 1 ) + j ) ,0 );
hidden.getListOfWeightIn().set( i, newWeight );
neuron.getListOfWeightIn().set( hidden_i, newWeight );
```

对于输出层：

```
newWeight = neuron.getListOfWeightOut().get(i) +
    delta.getValue( ( numberOfInputs + 1 ) *
            ( numberOfHiddenNeurons ) +
```

```
                ( i*(numberOfHiddenNeurons+1) )+j , 0);
neuron.getListOfWeightOut().set(i, newWeight);
```

3.7 实际应用——新生入学

在巴西，一个人进入大学的方法之一是参加考试，如果他/她达到了所申请课程需要的最低成绩，那么就可以入学。为了演示反向传播算法，让我们考虑下这个场景。表 3-8 展示的数据收集自一个大学数据库。第二列表示了人的性别（1 代表女性，0 代表男性）；第三列是归一化的成绩，最后一列由两个神经元组成（1,0 表示允许入学；0,1 表示放弃入学）。

表 3-8

样例	性别	分数	入学状态
1	1	0.73	1,0
2	1	0.81	1,0
3	1	0.86	1,0
4	0	0.65	1,0
5	0	0.45	1,0
6	1	0.70	0,1
7	0	0.51	0,1
8	1	0.89	0,1
9	1	0.79	0,1
10	0	0.54	0,1

图 3-16 展示了解决该问题的神经网络的架构。

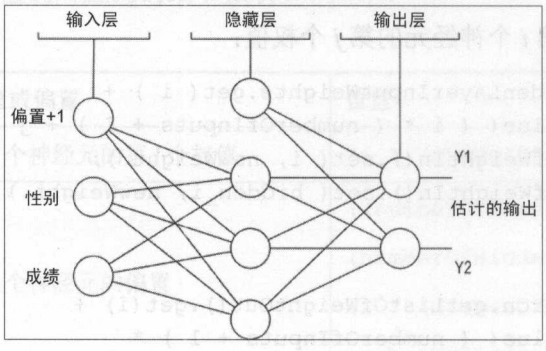


图 3-16

现在，让我们来分析 testBackpropagation()这个测试方法。如下：

```
private void testBackpropagation(){
    NeuralNet testNet = new NeuralNet();

    testNet = testNet.initNet(2, 1, 3, 2);

    System.out.println("---BACKPROPAGATION INIT NET---");

    testNet.printNet(testNet);

    NeuralNet trainedNet = new NeuralNet();

    // first column has BIAS
    testNet.setTrainSet(new double[][] { { 1.0, 1.0, 0.73 }, { 1.0, 1.0, 0.81 },
{ 1.0, 1.0, 0.86 }, { 1.0, 1.0, 0.95 }, { 1.0, 0.0, 0.45 }, { 1.0, 1.0, 0.70 },
{ 1.0, 0.0, 0.51 }, { 1.0, 1.0, 0.89 }, { 1.0, 1.0, 0.79 }, { 1.0, 0.0, 0.54 } });
    testNet.setRealMatrixOutputSet(new double[][] { {1.0, 0.0}, {1.0, 0.0},
{1.0, 0.0}, {1.0, 0.0}, {1.0, 0.0}, {0.0, 1.0}, {0.0, 1.0}, {0.0, 1.0}, {0.0,
1.0}, {0.0, 1.0} });
    testNet.setMaxEpochs(1000);
    testNet.setTargetError(0.002);
    testNet.setLearningRate(0.1);
    testNet.setTrainType(TrainingTypesENUM.BACKPROPAGATION);
    testNet.setActivationFnc(ActivationFncENUM.SIGLOG);

    testNet.setActivationFncOutputLayer(
            ActivationFncENUM.LINEAR );

    trainedNet = testNet.trainNet(testNet);

    System.out.println();
    System.out.println("---BACKPROPAGATION TRAINED NET---");

    testNet.printNet(trainedNet);
}
```

反向传播测试逻辑与有限线性元和感知机类似。首先，创建 NeuralNet 类的对象，然后用它初始化网络：输入层两个神经元，有一个隐藏层，里面有 3 个神经元，输出层有两个神经元。训练的数据来自于表 3-8。最大迭代次数设置为很大，因为反向传播算法延长了学习过程。最后，打印反向传播算法训练的网络权值和 MSE 集合。对训练结果的总结如图 3-17 所示。

```
---------BACKPROPAGATION TRAINED NET---------
### INPUT LAYER ###
Neuron #1:
Input Weights:
[0.16924330190980352]
Neuron #2:
Input Weights:
[0.3618079646279735]
Neuron #3:
Input Weights:
[0.7224791480888691]

### HIDDEN LAYER ###
Hidden Layer #1
Neuron #1
Input Weights:
[]
Output Weights:
[0.4151058895965083, 0.4724549355410933]
Neuron #2
Input Weights:
[-1.0244917646240137, -0.7571668108476453, -0.5562542919121999]
Output Weights:
[0.3767927510146706, 0.5529783319564584]
Neuron #3
Input Weights:
[-1.7341019854039823, -0.26680882030848974, 0.1312890017926201]
Output Weights:
[0.4069412654832925, -0.0995971490391232]
Neuron #4
Input Weights:
[-0.6010312196385755, -2.387731208398923, 0.02899942378387367]
Output Weights:
[-1.0251768985861485, 0.6448031463330727]

### OUTPUT LAYER ###
Neuron #1:
Output Weights:
[0.4658569521968512]
Neuron #2:
Output Weights:
[0.46127955517468777]
```

图 3-17

分析每次迭代的 MSE 绘制的图表，可以得出结论：神经网络在性别和成绩的基础上学习了分类，来判断一个人是否会进入该所大学，结果如图 3-18 所示。

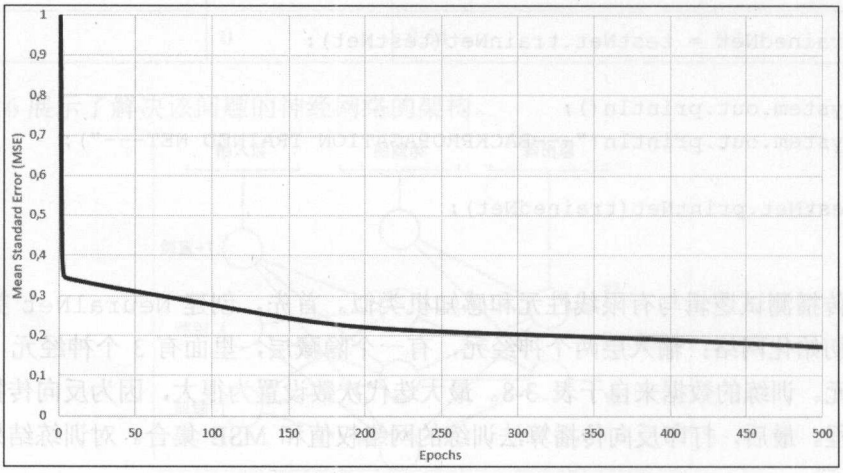


图 3-18

3.8 小结

在本章中，我们认识了神经网络如何被用来解决线性可分的问题并讨论了对非线性数据分类的局限性。为了克服这些局限性，我们展示了多层感知机（MLP）和一个新的学习算法：反向传播。我们还看到了 MLP 适用的问题类型，例如分类和回归。掌握这些概念对于理解它们在随后方法中的应用很重要。Java 代码实现探讨了反向传播算法对于更新输出层和隐藏层中权值的作用，并展示了一个实际应用来演示 MLP 和考虑问题的相应解决方法。

在下一章中，我们会探讨神经网络的其他学习范式——无监督学习，它与我们在本章中看到的学习算法略有不同，但是能产生神奇的结果。

第 4 章
自组织映射

本章介绍一种适合于无监督学习的神经网络结构：自组织映射（Self-Organizing Maps，SOMs）神经网络，也称为 Kohonen 网络。这种神经网络的特点是，在没有任何期望输出的情况下对数据记录进行分类。本章将继续探讨如何实现自组织映射网络，以及通过一个应用来证明其能力。本章的重点如下：

- 无监督学习神经网络
  - 竞争学习
- Kohonen 自组织映射神经网络
  - 一维 SOMs
  - 二维 SOMs
- 无监督学习解决的问题
- Kohonen 算法编程
- 实际问题

4.1 神经网络无监督学习方式

在第 2 章中，我们已经熟悉这种学习类型，现在将深入探讨这个学习范例的特征。无监督学习本质上旨在仅通过数据集本身所含信息找出数据模型。在这里，无监督学习算法会在没有任何误差度量的情况下调整参数（在神经网络中指权值），这是区分无监督学习和监督学习的关键特征。学习本身启发于神经学，相似的刺激会产生相似的反应。因此，将这种基本知识应用于人工神经网络，我们就可以说相似的输入数据产生相似的输出，并且

这些输出是可以聚类的。

虽然这种学习可以用于数学领域，例如统计学，但其核心功能是被设计专用于处理机器学习问题，如数据挖掘和模式识别。神经网络是机器学习的子领域，倘若它们的结构允许迭代学习，那么神经网络将是应用机器学习概念的一个很好的框架。

人们希望无监督学习算法用于，当对数据没有明确目标并且需要找出数据间隐藏模型的情况。大多数无监督学习应用的目标是聚类任务，这意味着类似的数据点聚集在一起，不同的数据点在不同的聚簇里。此外，无监督学习还适用于降维，其中包括希望在较小维度域中对一些多维数据进行分类或重新组织。在参考文献[Duda et. al,2001;Hinton et.al,1991;Rummerlhart & Zi 注 er;1985;Kohonen, 1982]中，读者可以找到一系列有用的文章，这些文章展示了更多关于无监督学习的应用示例。

4.2 无监督学习算法介绍

目前已有大量的无监督学习算法，这些算法不仅用于神经网络，如 K-means、最大期望算法、矩量法等。这些算法假设把整个数据集都作为将被学习的知识，所有学习算法的共同特点是它们没有当前数据集的输入—输出关系。然而，人们希望得到这些数据的不同含义，这就是所有无监督学习算法的目标。

将无监督学习这个概念引入神经网络，让我们看看人工神经网络以及它在无监督组织网络下怎么处理数据，如图 4-1 所示。

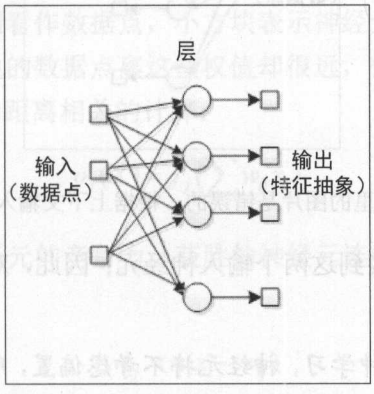


图 4-1

我们认为神经网络的输出是原始数据集的抽象。与监督学习范例相反，神经网络的输入数据集和数据变量之间并没有因果关系，反而我们希望神经网络能导出结果变量，这些

结果变量将能够对当前数据给出另一层含义。在监督学习算法中，输出的数量通常较少，然而对于非监督学习，需要产生抽象的数据表征，那就可能需要大量的输出。然而，除了分类任务，输出的含义完全不同于监督学习中对输出的含义。通常，每个输出神经元负责表示一个特征或者表示输入数据的一个类。在大多数神经网络体系结构中，并非所有输出神经元需要被同时激活，通常只有一组有限的输出神经元被激发，这意味着神经元能够更好地表示提供给神经网络输入数据的大部分信息。

提示：

无监督学习相对于监督学习的优势是，前者对于大数据集的学习需要消耗更少的计算能力，所以时间花费会线性增加，而对于监督学习，其时间花费则以指数方式增加。

在本章中，我们将探讨两种无监督学习算法：竞争学习和 Kohonen 自组织映射网络。

竞争学习或者赢家通吃

顾名思义，竞争学习处理输出神经元之间的竞争，以确定哪一个是胜者。为了便于理解，假设我们要训练具有两个输入和 4 个输出的单层神经网络，如图 4-2 所示。

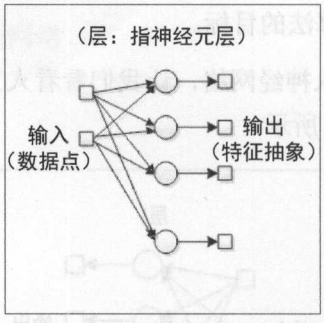


图 4-2（译者注：这里的图片是错误的，根据上下文输入神经元的个数为 2）

然后每个输出神经元连接到这两个输入神经元；因此，对于每个神经元，有两个权值。

提示：

对于这种学习，神经元将不考虑偏置，所以神经元只需处理加权的输入即可。

竞争在神经元处理完数据之后开始。获胜神经元将是产生最大输出值的神经元。与监督学习算法相比，另一个差异在于只有获胜神经元才可以更新其权值，而其他神经元则保

持不变，这就是所谓的“赢家通吃”规则。这意在让“更接近”输入的神经元赢得竞争。

鉴于所有的输入神经元 i 通过权值 ω_{ij} 连接所有的输出神经元 j。在我们的例子中，有一组权值：

$$W=\begin{bmatrix} w_{11} & w_{12} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \\ w_{14} & w_{24} \end{bmatrix}$$

倘若每个神经元的权值都具有和输入数据相同的维度，我们认为所有输入数据集合和所有神经元权值聚集在同一张图中，如图 4-3 所示。

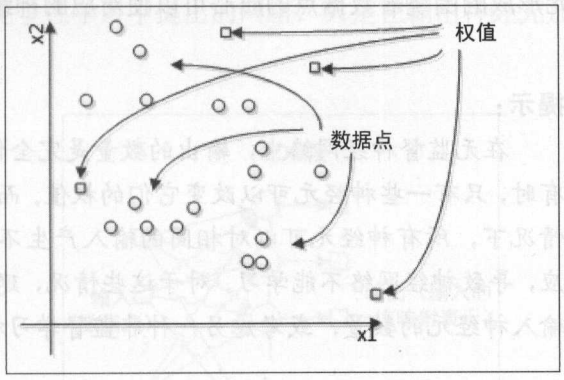


图 4-3

在这张图中，我们把圆圈看作数据点，小方块表示神经元权值。可以看出一些数据点离一些权值方块很近，而其他的数据点离这些权值却很远，但离其他的权值却很近。神经网络执行与输入和权值之间的距离相关的计算：

$$o_j(X)=f_j(\sum w_{ij}x_i)$$

这个等式的结果反映神经元的竞争力。获胜的神经元连接将会通过以下更新规则调整神经元：

$$\Delta w_{ij}=\eta(x_i-w_{ij})$$

其中 η 表示学习率，经过多次迭代后，权值趋近于一个数据点，该数据点在对应神经元能触发最大输出值，该权值更新要么太小，要么呈锯齿状下降，最终，训练好神经网络后，图像呈现另一种形状，如图 4-4 所示。

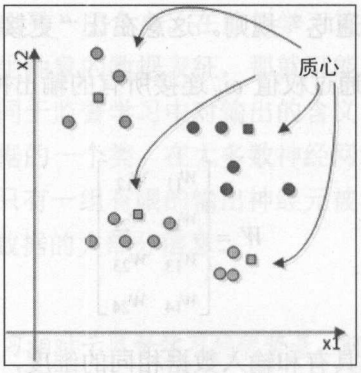


图 4-4

可以看出，神经元形成的围绕着数据点的质心可以使对应的神经元比它的竞争者更具竞争力。

提示：

在无监督神经网络中，输出的数量是完全任意的，有时，只有一些神经元可以改变它们的权值。而在其他情况下，所有神经元可以对相同的输入产生不同的反应，导致神经网络不能学习。对于这些情况，建议审查输入神经元的数量，或考虑另一种非监督学习模型。

竞争学习基本上有两个停止条件。

- 预定义迭代次数：防止我们的算法运行相对较长的时间而没有收敛。
- 最小权值更新：防止算法运行超过必要时间。

4.3 Kohonen 自组织映射

这种网络架构是由芬兰教授 Teuvo Kohonen 在 20 世纪 80 年代初提出的。它由一个能够提供一维或二维数据“可视化”的单层神经网络构成。

提示：

理论上，Kohonen 网络将能够表示三维或者更高维的数据；然而，例如在本书中，就不能在不重叠某些数据的情况下显示三维图表。因此，在本书中，我们将仅处理一维和二维 Kohonen 网络。

Kohonen 自组织映射和传统的单层竞争神经网络之间的主要区别是邻近神经元的概念。在神经网络中，通常神经元在输出层的位置顺序是不重要的，但是在自组织映射（SOM）中，邻近神经元在学习过程中扮演相当重要的角色。

SOM 有两种功能：映射和学习，在映射模式中，输入数据被分类在最合适的神经元中。而在学习模式中，输入数据帮助学习算法构建“映射”，该映射可以理解为来自特定数据集的低维表示。

本章将介绍两种 SOM，分别是一维 SOM 和二维 SOM。

4.3.1 一维 SOM

这种架构类似于竞争性学习中提出的网络，只是在输出神经元加入了临近神经元，如图 4-5 所示。

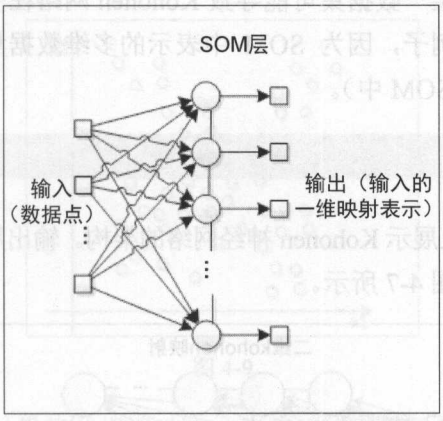


图 4-5

可以注意到每个输出层的神经元都有两个临近神经元。类似地，产生最大值的神经元更新它的权值，但在 SOM 中，邻近神经元也会以相对较低的学习率来更新它们的权值。

假设所有的输出神经元都遵循同一个结构（如一维情况下是一条线），那么邻近神经元的效果将把激活区域传递到更广的映射范围。邻近神经元功能还允许我们更好地探索输入空间的属性，因为它迫使神经网络维持神经元间的连接，因此除了仅形成的簇之外还会产生更多的信息。

图 4-6 包含带神经权值的输入数据点，我们可以看到神经元形成的路径。

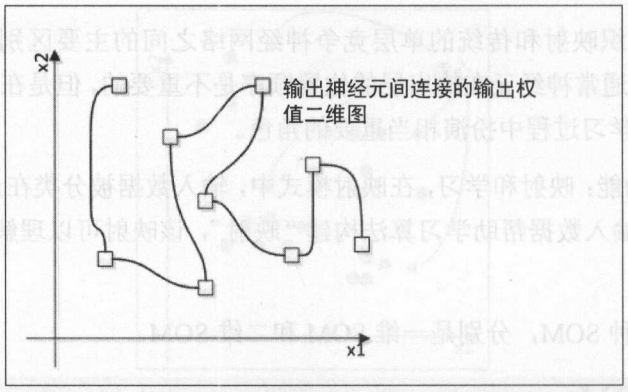


图 4-6

在图 4-6 中，为了简便，我们只绘制了输出权值，以演示如何在（在这个例子中）二维空间中设计映射。经过多次迭代训练后，神经网络收敛到表示所有数据点的最终形状。运用这种神经网络结构，某一数据集可能导致 Kohonen 网络在空间上产生另一个形状。这是实现降维的一个很好的例子，因为 SOM 中表示的多维数据集能够产生“概述”整个数据集的单独的线（在一维 SOM 中）。

4.3.2 二维 SOM

这是最常用于在视觉上展示 Kohonen 神经网络的架构。输出层是包含 $N \times N$ 个神经元的矩阵，像网格一样互连，如图 4-7 所示。

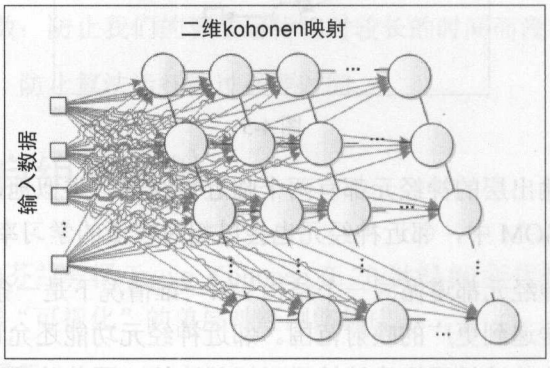


图 4-7

在二维 SOM 中，每个神经元现在将具有多达 4 个相邻神经元，尽管在一些表示中，对角神经元也可以考虑，因此每个神经元导致多达 8 个邻居。基本上，二维 SOM 的工作原理是相同的。让我们看一个 3×3 的 SOM 图在二维图表中的示例（两个输入变量），如图 4-8 所示。

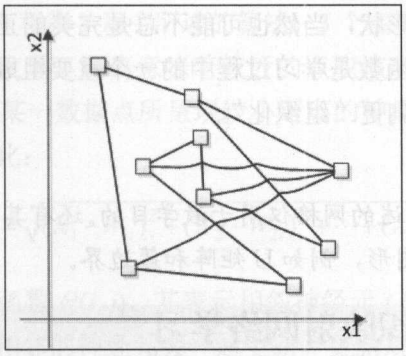


图 4-8

首先，未经训练的 Kohonen 网络显示出一个非常奇怪和扭曲的形状。权值的形状仅取决于将要传给 SOM 的输入数据。让我们来看一个映射如何开始自组织的例子。

- 假设有如图 4-9 所示的密集数据集。

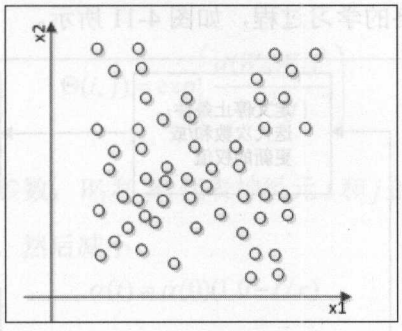


图 4-9

- 应用 SOM 之后，二维形状慢慢改变，直到网络取得最后的形状，如图 4-10 所示。

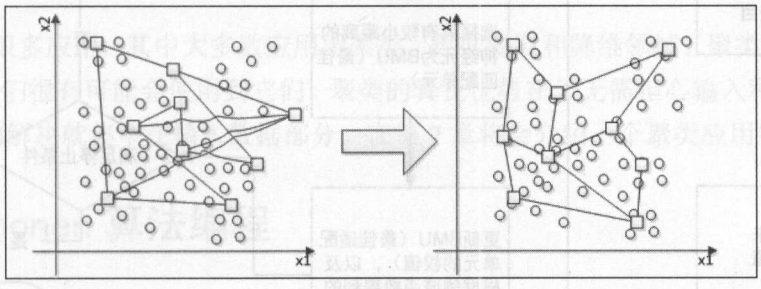


图 4-10

最终的二维 SOM 的形状不一定是完美的正方形，而可能和刚从数据集画出的形状相似。邻域函数是学习过程中的一个重要组成，因为它接近于图形中的邻近神经元，使结构变成

更有序的二维 SOM 的最终形状，当然也可能不总是完美的正方形；相反，它将类似于从数据集中绘制的形状。邻域函数是学习过程中的一个重要组成部分，因为它接近绘图中的相邻神经元，并且使结构变到更“组织化”。

提示：

上文所述的网格仅用于教学目的。还有其他的方式说明 SOM 图形，例如 U 矩阵和簇边界。

4.3.3 逐步实现自组织映射网络学习

SOM 旨在通过聚集在输出上触发相同响应的数据点来对输入数据进行分类。最初，未训练的网络将产生随机输出，但是当呈现更多的示例时，神经网络识别应该更经常地激活哪些神经元，然后，它们在 SOM 输出空间中的“位置”被改变。该算法是基于竞争学习的，这意味着获胜神经元（也称为最佳匹配神经元，BMU）将会更新它的权值和临近神经元的权值。

流程图显示了 SOM 网络的学习过程，如图 4-11 所示。

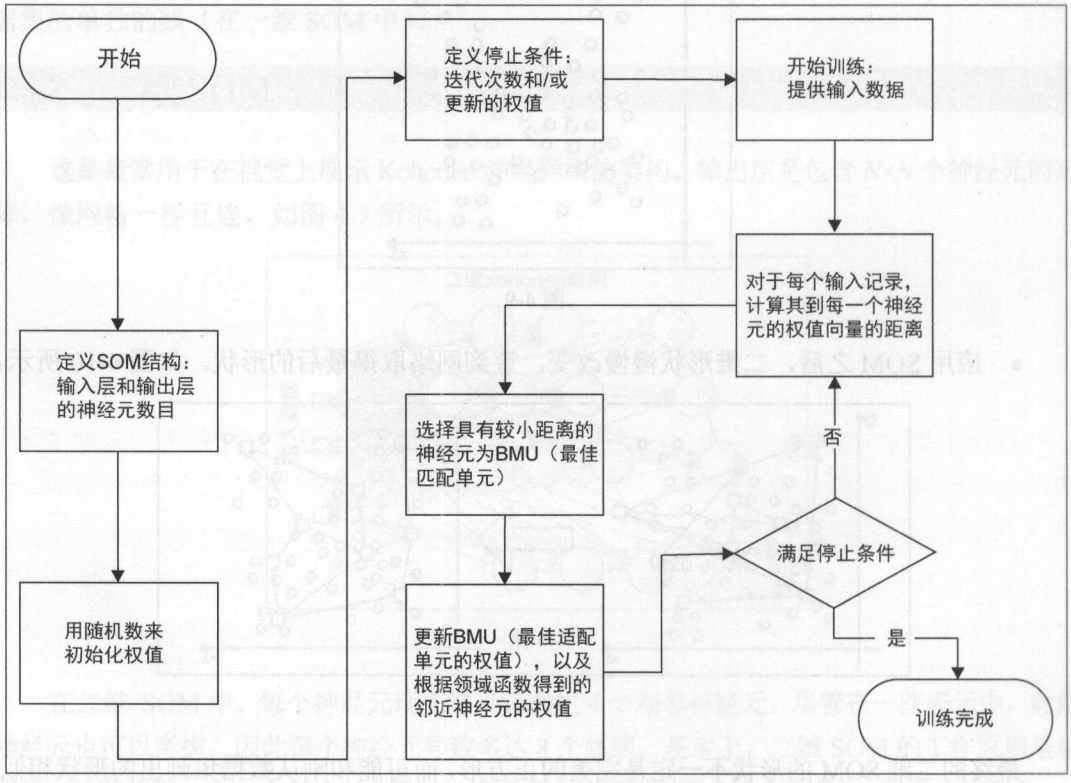


图 4-11

此学习有点类似于第 2 章和第 3 章中所述的算法。3 个主要差异是利用距离决定 BMU、权值更新规则以及缺少误差度量。距离意味着较近的点应该产生类似的输出。因此，在这里，确定 BMU 的标准是到某一数据点所呈现较小距离的神经元。通常使用欧氏距离，在本书中，我们将使用它来简化：

$$d(X,W_i)=\sqrt{(X_1-W_{1i})^2+(X_2-W_{2i})^2+\cdots+(X_n-W_{ni})^2}$$

权值更新规则使用邻域函数 $\Theta(i,j)$，其表示相邻神经元 i 离神经元 j 有多近。记住，在 SOM 中，BMU 神经元与其相邻的神经元会一起更新。更新规则如下：

$$\Delta W_{kj}=\Theta(i,j)\alpha(X_k-W_{kj})$$

其中 α 表示学习率；Θ 表示邻域函数；X_k 表示第 k 个输入；W_{kj} 是连接第 k 个输入到第 j 个输出的权值。该学习算法的另一个特征是学习速率和邻域函数都取决于迭代次数。邻域函数通常是高斯函数：

$$\Theta(i,j)=\exp\left(\frac{d(W_i,W_j)^2}{2\sigma^2(t)}\right)$$

其中 σ 表示方差的高斯参数，W_i 和 W_j 代表神经元 i 和 j 的权值，t 表示迭代次数。

学习速率从初始值开始，然后减小：

$$\alpha(t)=\alpha(0)(1.0-t/r)$$

其中 r 代表学习率的一个参数。

4.3.4 如何使用 SOM

SOM 有很多应用，其中大多数应用于聚类、数据提取和降维领域。聚类应用是最受关注的，因为人们很有可能会使用到它们。聚类的真正优势在于无需担心输入和输出的关系，当然，问题的解决就集中在输入数据部分。在第 7 章将会介绍一个聚类应用案例。

4.4 Kohonen 算法编程

现在，是时候亲自动手用 Java 实现 Kohonen 神经网络了。基于之前 Java 代码的变化以及应用面向对象编程概念，我们可以轻易实现新特征，且无需重写项目中已经完成的代码。为了简便，现在我们只实现竞争学习和单神经元权值更新规则，所做的更改如表 4-1 所示。

表 4-1

类名：NeuralNet	
注意：这个类在先前版本中已存在，更新如下	
属性	
`private double[][] validationSet;`	存储输入数据的验证集的矩阵
方法	
注意：此属性的 getters 和 setters 方法也已创建	
Java 实现类：文件 NeuralNet.java	
接口名称：Validation	
提示：在 Java 中，接口是可能具有常量属性和/或方法签名的结构，必须在类中实现	
属性	
None	
方法	
`public void netValidation (NeuralNetn);`	执行神经网络验证，在 PC 屏幕上打印一些结果
	参数：`NeuralNet` 对象(训练的神经网络)
	返回：-
java 实现接口：Validation.java	
类名：Kohonen	
注意：此类继承自 NeuralNet 并实现验证接口。	
属性	
None	
方法	
`public NeuralNet train (NeuralNet n)`	通过应用 Kohonen 算法训练神经网络，此方法覆盖 `Training` 类中的方法
	参数：`NeuralNet` 对象（训练的神经网络）
	返：`NeuralNet` 类（通过 Kohonen 方法训练的神经网络）

续表

方法	
private NeuralNet initNet (NeuralNet n)	用 0 初始化来自输入层的神经元列表 listOfWeightOut
	参数：未初始化输入层的 NeuralNet 对象
	返回：初始化输入层的 NeuralNet 对象
private ArrayList<Double> calcEuclideanDistance (NeuralNet n, double[][] data, int row)	计算训练数据和神经网络权值之间的欧氏距离
	参数：NeuralNet 对象、训练数据、训练数据的行数
	返回：欧氏距离距离实值列表
private NeuralNet fixWinnerWeights (NeuralNet n, int winnerNeuron, int trainSetRow)	调整优胜者神经元的权值（基于欧几里德距离列表）
	参数：NeuralNet 对象、获胜神经元的索引、训练集行数
	返回：修改了输入层权值的 NeuralNet 类
public void netValidation(NeuralNet n)	调整优胜者神经元的权值（基于欧几里德距离列表）
	参数：训练完的神经网络 NeuralNet 对象
	返回：-
java 实现类：Kohonen.java	

类图更改如图 4-12 所示，前面章节中已经解释的属性和方法及其配置方法（getter 和 setter）并未显示在其中。

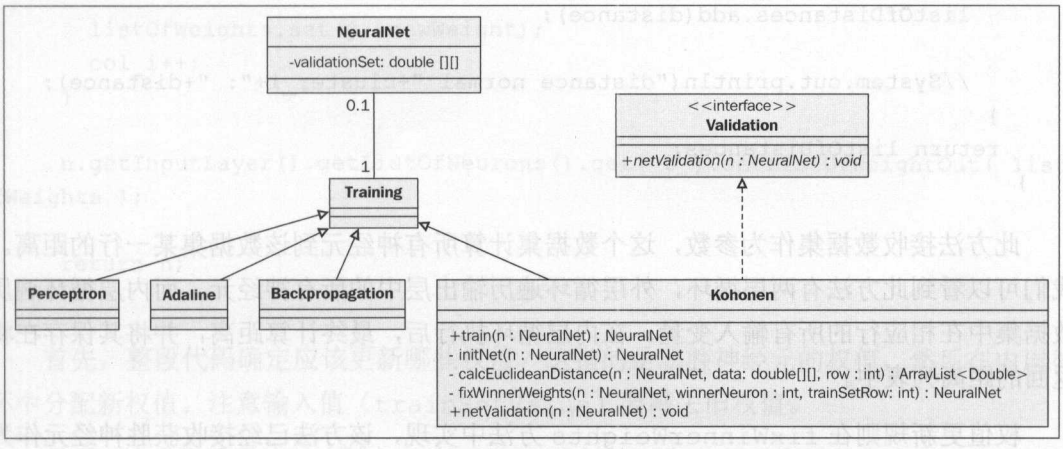


图 4-12

4.4.1 探索 Kohonen 类

Kohonen 类实现验证接口，提供验证方法以确保选择正确的输出神经元。让我们重点关注这个类中的 3 个关键方法：calcEuclideanDistance、fixWinnerWeights 和 train。

根据 SOM 学习算法中所示的等式计算欧氏距离，如以下代码所示：

```
    private ArrayList<Double> calcEuclideanDistance(NeuralNet n, double[][]
data, int row) {
      ArrayList<Double> listOfDistances = new ArrayList<Double>();

      int weight_i = 0;
      for(int cluster_i = 0; cluster_i < n.getOutputLayer().
getNumberOfNeuronsInLayer(); cluster_i++) {

        double distance = 0.0;

        for(int input_j = 0; input_j < n.getInputLayer().
getNumberOfNeuronsInLayer(); input_j++) {

          double weight = n.getInputLayer().getListOfNeurons().get(0).
getListOfWeightOut().get(weight_i);
          distance = distance + Math.pow(data[row][input_j] - weight, 2.0);
          weight_i++;

        }

        listOfDistances.add(distance);

        //System.out.println("distance normal "+cluster_i+": "+distance);
      }
      return listOfDistances;
    }
```

此方法接收数据集作为参数，这个数据集计算所有神经元到该数据集某一行的距离。我们可以看到此方法有两层循环，外层循环遍历输出层中的所有神经元，而内层循环遍历数据集中在相应行的所有输入变量。在内层循环执行后，最终计算距离，并将其保存在将返回的距离列表中。

权值更新规则在 fixWinnerWeights 方法中实现，该方法已经接收获胜神经元作为参数。该方法的代码如下：

```
    private NeuralNet fixWinnerWeights(NeuralNet n, int winnerNeuron, int
trainSetRow) {
      int start, last;

      start = winnerNeuron * n.getInputLayer().getNumberOfNeuronsInLayer();

      if(start < 0) {
        start = 0;
      }

      last = start + n.getInputLayer().getNumberOfNeuronsInLayer();

      List<Double> listOfOldWeights = new ArrayList<Double>();
      listOfOldWeights = n.getInputLayer().getListOfNeurons().get( 0 ).
getListOfWeightOut().subList(start, last);

      ArrayList<Double> listOfWeights = new ArrayList<Double>();
      listOfWeights = n.getInputLayer().getListOfNeurons().get( 0 ).
getListOfWeightOut();

      int col_i = 0;
      for (int j = start; j < last; j++) {
        double trainSetValue = n.getTrainSet()[trainSetRow][col_i];
        double newWeight = listOfOldWeights.get(col_i) +
            n.getLearningRate() *
            (trainSetValue - listOfOldWeights.get(col_i));

        //System.out.println("newWeight: " + newWeight);

        listOfWeights.set(j, newWeight);
        col_i++;
      }

      n.getInputLayer().getListOfNeurons().get( 0 ).setListOfWeightOut( list
OfWeights );

      return n;
    }
```

首先，整段代码确定应该更新哪些权值，这指的是获胜神经元的权值。然后在内层循环中分配新权值。注意输入值（trainSetValue）要减去旧权值。

最后，让我们看看这些方法如何在 Train 方法中一起使用。为了节省空间，我们只关

注迭代循环部分：

```
for (int epoch = 0; epoch < n.getMaxEpochs(); epoch++) {

  //System.out.println("### EPOCH: "+epoch);

  for (int row_i = 0; row_i < rows; row_i++) {
    listOfDistances = calcEuclideanDistance(n, trainData, row_i);

    int winnerNeuron = listOfDistances.indexOf(Collections.
min(listOfDistances));

    n = fixWinnerWeights(n, winnerNeuron, row_i);
  }
}
```

对于训练集中的每一行，使用欧氏距离来计算距离，并且紧接着确定获胜神经元。然后更新权值，并且学习过程进入下一次迭代。

4.4.2 Kohonen 实现（动物聚类）

在本节中，我们将在实践中解释 Kohonen 算法。想象一下，有一些动物，它们的 3 个特点是：有皮毛（是/否）、陆地生物（是/否）、有乳腺（是/否）。我们的目标是将动物聚类到两个未知的组中，表 4-2 总结了以下数据。

表 4-2

#	动物	有皮毛(有=1 / 没有 =−1)	陆地生物(是 =1 / 不是 =−1)	有乳腺(有 =1 / 没有=−1)
1	蝙蝠	1	−1	1
2	鲨鱼	−1	−1	−1
3	海象	−1	−1	1
4	蜘蛛	1	1	−1
5	河马	−1	1	1
6	苍蝇	1	−1	−1
7	毒蛇	−1	1	−1
8	猴子	1	1	1

图 4-13 展示了用于解决这个问题的 Kohonen 神经网络的架构。

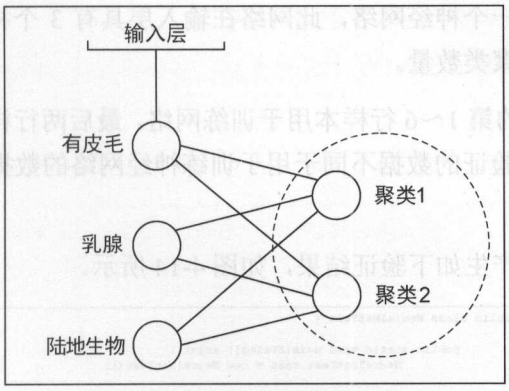


图 4-13

接下来，我们分析测试方法 testKohonen()。如下所示：

```
private void testKohonen(){
    NeuralNet testNet = new NeuralNet();

    //2 inputs because "bias"
    testNet = testNet.initNet(2, 0, 0, 2);

    NeuralNet trainedNet = new NeuralNet();

    testNet.setTrainSet(new double[][] { { 1.0, -1.0, 1.0 }, { -1.0, -1.0,
-1.0 }, { -1.0, -1.0, 1.0 }, { 1.0, 1.0, -1.0 }, { -1.0, 1.0, 1.0 }, { 1.0, -1.0,
-1.0 } });

    //viper and monkey, respectively:
    testNet.setValidationSet(new double[][] { {-1.0, 1.0, -1.0}, {1.0, 1.0,
1.0} } );

    testNet.setMaxEpochs(10);
    testNet.setLearningRate(0.1);
    testNet.setTrainType(TrainingTypesENUM.KOHONEN);

    trainedNet = testNet.trainNet(testNet);

    System.out.println();
    System.out.println("---------KOHONEN VALIDATION NET---------");

testNet.netValidation(trainedNet);
```

Kohonen 测试逻辑遵循与之前实现中使用相同的步骤。首先，创建 NeuralNet 类的

一个对象，用于初始化一个神经网络，此网络在输入层具有 3 个神经元，在输出层有两个神经元，表示要实现的聚类数量。

之后，前面表格中的第 1～6 行样本用于训练网络，最后两行样本数据用于验证神经网络。重要的是确保用于验证的数据不同于用于训练神经网络的数据。最后，调用训练神经网络的方法。

测试实例结束时会产生如下验证结果，如图 4-14 所示。

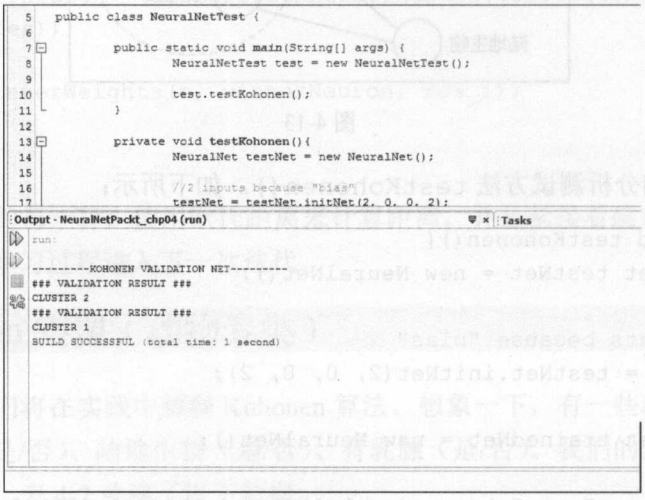


图 4-14

通过分析验证结果，我们发现神经网络可以聚类两个不同种类的动物。

- **聚类 1**：哺乳动物（猴子）。
- **聚类 2**：非哺乳动物（毒蛇）。

4.5 小结

在本章中，我们已经了解了如何将非监督学习算法应用于神经网络。我们已经为此介绍了一个新的合适的网络结构，即 Kohonen 的 SOM。此外，无监督学习已被证明与监督学习方法一样强大，因为它仅集中在输入数据上，而不需要输入—输出映射关系。我们已经看到了两种新的训练算法：竞争学习及其对 Kohonen 网络的扩展。除了提供大数据集的图形表示之外，SOM 还在聚类和维数降低中起作用。以到目前为止所学到的内容，我们可以转入下一章的学习，讨论有趣的天气预测实际应用。

第 5 章
天气预测

本章介绍了神经网络在未来天气数据预测方面的应用，我们将全面了解整个从神经网络设计到将其应用于天气预测的过程、如何选择神经结构、神经元的数量以及如何选择和预处理数据。然后，将提供给读者神经网络预测天气变量所用的数据集，并且这个数据集可适用于 Java 语言编程。本章涵盖的主题如下：

- 用于预测问题的神经网络
- 数据选择
  - 输入/输出变量
  - 过滤
- 预处理
  - 标准化
- Java 实现
  - 改进
- 神经网络经验设计

5.1 针对预测问题的神经网络

到目前为止，读者已经了解许多神经网络的实现和架构，现在也是时候接触更复杂的情况了。神经网络在预测领域的力量确实是令人惊讶的，因为它们可以从历史数据中习得一种模型，在这个模型中，神经连接适用于根据一些输入数据产生相同结果。例如，给定一种境况（原因），必有一种结论（结果），这就被编码成数据。神经网络可用于学习从境

况到结论（或从原因到结论）映射的非线性函数。

预测问题是神经网络应用的一个有趣的类别。让我们来看看包含天气数据的样本表（见表 5-1）。

表 5-1

日期	平均温度	气压	湿度	降水量	风速
July 31	23° C	880 mbar	66%	16 mm	5 m/s
August 1	22° C	881 mbar	78%	3 mm	3 m/s
August 2	25° C	884 mbar	65%	0 mm	4 m/s
August 3	27° C	882 mbar	53%	0 mm	3 m/s
…					
December 11	32° C	890 mbar	64%	0 mm	2 m/s

表 5-1 描述了 5 个变量，其包含从假定城市收集的天气数据的假设值，仅用于本示例。现在，让我们假设每个变量包含一系列随时间顺序取得的值。我们可以把每一列看作一个时间序列。在时间序列图上，可以看到天气数据随时间的变化，如图 5-1 所示。

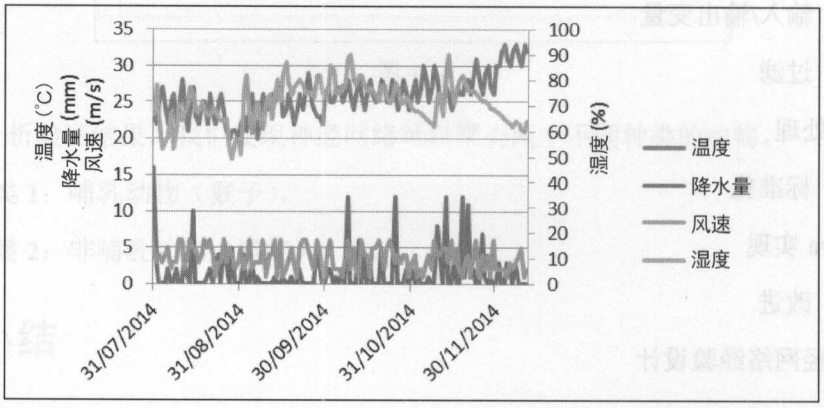


图 5-1

这些时间序列之间的关系显示某个城市天气的动态表示，如图 5-1 所示。我们确实希望神经网络学习这些动态；然而，又有必要更多地了解这一现象，因为我们需要以神经网络可以处理它的方式来构造这些数据。

只有在构造完数据之后，我们才能构造神经网络，即输入层、输出层、隐藏层神经元个数。存在可能适合于预测问题的许多其他架构，例如径向基函数神经网络和反馈神经网

络。在本章中，我们将通过反向传播学习算法来处理前馈多层感知器网络，以演示如何可以简便地利用这种架构来预测天气变量。此外，该架构在数据选择良好的情况下呈现非常好的泛化结果，并且在设计过程中涉及很少的复杂性。

设计用于预测处理的神经网络的全部流程，如图 5-2 所示。

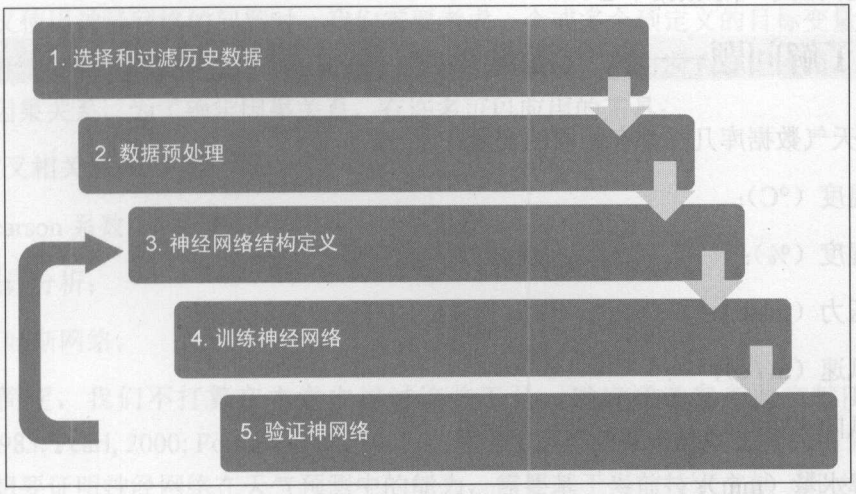


图 5-2

如果神经网络验证失败（步骤 5），则通常会定义新的结构（步骤 3），但往往可以重复步骤 1 和 2。图中的每个步骤将在本章的以下部分讨论。

5.2 无数据，无神经网络——选择数据

首先要做的事是选择适当的相关数据，这些数据携带了多数我们希望神经网络重现的系统动态信息。在例子中，我们需要选择与天气预报相关的数据。

提示：

在选择数据时，获得专家对于过程及其变量的意见是非常有用的。专家的确能够极大程度地帮助我们理解变量之间的关系，从而以适当的方式选择它们。

在本章中，我们将使用巴西气象研究所的数据，该数据可免费在互联网上获取，我们有权将其应用于本书。不管怎样，读者可以在开发应用时使用来自互联网的任何免费的天气数据库。一些英文的天气数据库示例如下：

- Wunderground；
- Open weather map；
- Yahoo weather API；
- 美国国家气象数据中心。

5.2.1 了解问题——天气变量

任何天气数据库几乎都有相同的变量：

- 温度（°C）；
- 湿度（%）；
- 压力（mbar）；
- 风速（m / s）；
- 风向（°）；
- 降水量（mm）；
- 日照（h）；
- 太阳能（W / m^2）。

上述这些数据通常是由气象站、卫星或雷达按每小时或每天的频率收集的。

提示：

根据数据收集频率，一些变量可以用平均值、最小值或最大值进行汇总。数据单元可能在不同数据源间有所差异，这就是为什么要一直监测这些数据单元。

5.2.2 选择输入输出变量

神经网络作为一个非线性模块，可能有预定义数量的输入和输出，所以我们必须选择每个天气变量在这个应用中将扮演的角色。换句话说，我们必须选择神经网络要预测的变量以及通过使用哪些输入变量来进行预测。

提示：

关于时间序列变量，可以通过应用历史数据来导出新变量。这意味着，给定某个日期，可以考虑该日期的值和从过去日期收集（或概括）的数据，以此扩展变量的数量。

在定义使用神经网络的问题时，我们需要考虑一个或多个预定义的目标变量：预测温度、预测降水、测量日照等。然而，在某些情况下，可能需要对所有变量建模，并确定它们之间的因果关系。为了确定因果关系，有许多可以应用的工具：

- 交叉相关法；
- Pearson 系数；
- 统计分析；
- 贝叶斯网络。

为了简便，我们不打算在本章中探讨这些工具，建议读者参考此文献[Dowdy & Wearden,1983; Pearl, 2000; Fortuna et al., 2007]，可以了解更多关于这些工具的细节。相反，由于我们想要证明神经网络在天气预测中的能力，需要基于当前技术文献（即之前引用的参考文献），并根据其他 4 个变量来选择给定日期的平均温度。

5.2.3 移除无关行为——数据过滤

有时，从某些来源获取数据时会遇到一些问题。常见的问题如下：

- 某一记录和变量中缺少数据；
- 测量错误（例如，当一个值被严重标记时）；
- 异常值（例如，当一个值远远超出通常范围时）。

要处理这些问题，需要对所选数据执行过滤。神经网络将准确地重现与被训练的数据完全相同的动态变化，因此向神经网络提供数据是必须小心谨慎，以防不良数据。通常，要从数据集中删除包含不良数据的记录，确保只有“好”数据送到网络。

为了更好地理解过滤，让我们将数据集视为一个包含 n 个测量值和 m 个变量的大矩阵。

$$A=\begin{bmatrix} a_1(1) & \cdots & a_m(1) \\ a_1(2) & \cdots & a_m(2) \\ \vdots & \ddots & \vdots \\ a_1(n) & \cdots & a_m(n) \end{bmatrix}$$

其中 $a_j(i)$表示 i 时刻 j 变量的测量值。

所以，我们的任务是找到坏记录并删除它们。在数学上，有许多方法来识别坏记录。对于误差测量和异常值检测，以下 three-sigma 规则非常好：

$$|d_i| - \left|\frac{x_i - E[X]}{\sigma_X}\right| - \begin{cases} > 3 bad\ record, remove \\ \leqslant 3 good\ record, keep\ it \end{cases}$$

其中 x_i 表示第 i 次测量的值，$E[X]$表示平均值，σ_X 表示标准偏差，d_i 表示距平均值的加权距离。如果第 i 次测量的绝对距离未能满足小于 3 个记录，则第 i 次测量将被标记为不良测量，并且尽管来自同一实例（矩阵的行）的其他变量是良好的，但是数据集的整行都应该丢弃。

5.3 调整数值——数据预处理

从数据源收集的原始数据通常呈现不同的特性，例如数据范围、采样和类别。一些变量由其他测量值生成，而其他变量是通过总结或计算产生的，甚至是通过计算产生的。预处理意味着使这些变量的值适应于形成可以正确处理它们的神经网络。

关于天气变量，让我们来看看它们的范围、采样和类型，如表 5-2 所示。

表 5-2

变量	单位	范围	采样	类型
平均温度	°C	23.86–29.25	每小时	每小时测量的平均值
降水量	Mm	0–161.20	每天	每日降雨量累积和
日照	h	0–10.40	每天	接受太阳辐射的小时计数
平均湿度	%	65.50–96.00	每小时	每小时测量的平均值
平均风速	km/h	0.00–3.27	每小时	每小时测量的平均值

除了日照和降水量，其他变量都是通过测量产生的并且享有相同的采样。但是如果我们想要使用（例如每小时数据集），就必须预处理所有变量以使用相同的采样率。3 个变量都使用日平均值，但如果我们愿意，也可以使用每小时数据测量。然而，其范围肯定会更大。

平衡数据——标准化

标准化是把所有变量转化到相同数据范围（通常具有在 0 和 1，或−1 和 1 之间的较小值）的过程。这有助于神经网络在激活函数中呈现变量范围内的值，例如 S 形或双曲线正切函数，

如图 5-3 所示。

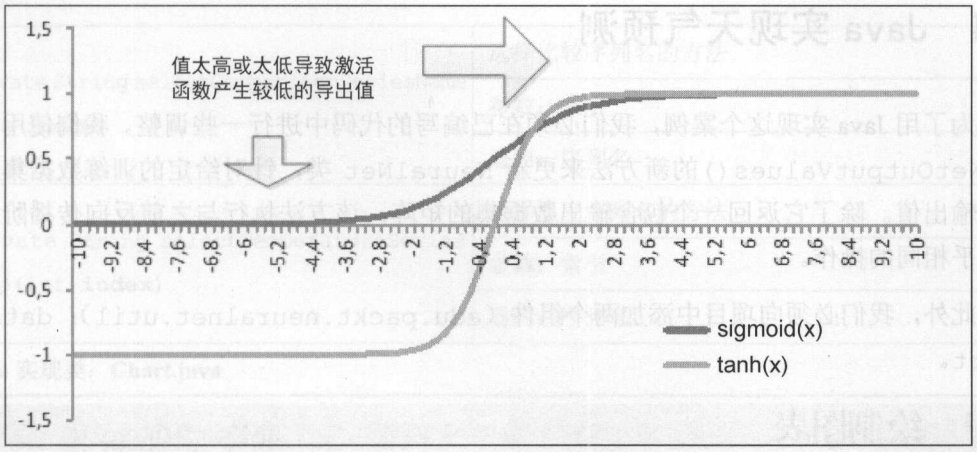


图 5-3

值太高或太低可能使神经元产生太高或太低的值，并且对激活函数也一样，因此导致这些神经元的导数太小，接近于零。

标准化应考虑数据集的预定义范围。执行如下：

$$X_{norm} = (N_{max} - N_{min})\left[\frac{X - X_{min}}{(X_{max} - X_{min})}\right] + N_{min}$$

其中 N_{min} 和 N_{max} 分别表示标准化的最小和最大极限；N_{min} 和 N_{max} 分别表示 X 变量的最小和最大限制；X 表示原始值，并且 X_{norm} 是指标准化值。如果我们想要标准化在 0 和 1 之间，例如，该等式简化如下：

$$X_{norm} = \frac{(X - X_{min})}{(X_{max} - X_{min})}$$

通过应用标准化，产生新的“标准化”数据集，并将其馈送到神经网络。还应该考虑到，馈送有标准化值的神经网络将被训练，其在输出端产生的值也是标准化的，因此逆（去标准化）过程也变得有必要。

$$X = (X_{max} - X_{min})\left[\frac{(X_{norm} - N_{min})}{(N_{max} - N_{min})}\right] + X_{min}$$

或者

$$X = (X_{max} - X_{min})[X_{norm}] + X_{min}$$

用于0和1之间的标准化。

5.4 Java实现天气预测

为了用Java实现这个案例，我们必须在已编写的代码中进行一些调整。我们使用名为getNetOutputValues()的新方法来更新NeuralNet类，针对给定的训练数据集产生一些输出值。除了它返回一个包含输出数据集的矩阵，该方法执行与之前反向传播阶段方法几乎相同的操作。

此外，我们必须向项目中添加两个组件（edu.packt.neuralnet.util）：data和chart。

5.4.1 绘制图表

图表可以使用免费提供的包JFreeChart（http://www.jfree.org/jfreechart/）在Java中绘制，此包附带本章的源代码。所以，我们设计了一个叫Chart的类。它基本上实现通过调用JFreeChart类的原生方法来绘制数据序列的方法。表5-3显示了此类中包含的一系列方法。

表5-3

类名：**Chart**	
属性	
public enum ChartPlotTypeENUM { FULL_DATA, COMPARISON; }	存储可以绘制的图表类型的枚举
方法	
public void plotXYData(Object[] vector, String **chartTitle**, String **xAxisLabel**, String **yAxisLabel**)	基于数据向量绘制*XY*图表的方法
	参数：要绘图的数据向量、图表标题、*X*轴标签、*Y*轴标签
	返回：-
public void plotXYData(double[][] **matrix**, String **chartTitle**,String **xAxisLabel**, String **yAxisLabel**, ChartPlotTypeENUM **chartPlotType**)	基于数据向量绘制XY图表的方法
	参数：要绘图的数据向量、图表标题、*X*轴标签、*Y*轴标签、绘制类型
	返回：-

续表

方法	
private String selectComparisonSeriesName (int **index**)	选择比较序列名的方法
	参数：索引
	返回：序列名
private String selectTemperatureSeries Name(int **index**)	选择温度序列名的方法
	参数：索引
	返回：序列名
java 实现类：Chart.java	

5.4.2 处理数据文件

要使用数据文件，我们必须实现一个名为 Data 的类。它目前从所谓的 csv 格式执行读取，这适合于数据导入和导出。此类还通过标准化对数据执行预处理，如表 5-4 所示。

表 5-4

类名：**Data**	
属性	
private String path;	存储 csv 文件目录路径的变量
private String fileName;	存储 csv 文件名称的（包括扩展名）的属性
public enum NormalizationTypesENUM { MAX_MIN, MAX_MIN_EQUALIZED; }	存储可以使用的标准化类型的枚举
构造函数	
public Data(String path, String fileName)	设置 path 和 filename 属性的构造函数
public Data( )	创建一个空对象的构造函数
方法	
注意：此属性的 getter 和 setter 方法也已创建	

续表

方法	
public double[][] rawData2Matrix(Data r) throws IOException	读取原始数据（csv 文件）并转换为 double 类型矩阵的方法
	参数：Data 对象
	返回：原始数据的 double 类型矩阵
private String defineAbsoluteFilePath (Data r) throws IOException	定义 csv 文件绝对路径的方法
	参数：Data 对象
	返回：csv 文件对接路径的字符串
public double[][] normalize(double[][] rawMatrix, NormalizationTypesENUM normType)	原始数据矩阵标准化的方法
	参数：double 类型原始数据矩阵、标准化类型
	返回：标准化的 double 类型矩阵
public double[][] denormalize(double[][] rawMatrix, double[][] matrixNorm, NormalizationTypesENUM normType	原始数据矩阵去标准化的方法
	参数：double 类型原始数据矩阵、double 类型标准化矩阵、标准化类型
	返回：去标准化的 double 类型矩阵
public double[][] joinArrays(ArrayList <double[][]> listOfArraysToJoin)	把数组（向量）加入矩阵的方法
	参数：数组列表
	返回：double 类型矩阵
Java 实现类：Data.java	

5.4.3 构建天气预测神经网络

为了预测天气，我们收集了来自巴西气象研究所（INMET 网站）的日常数据。数据来自位于亚马逊地区的巴西城市。

从 INMET 网站提供的 8 个变量中，选择了 5 个用于该项目，其中最大和最小温度的平均值变为平均温度变量。训练神经网络来预测平均温度，因此，神经网络的结构如图 5-4 所示。

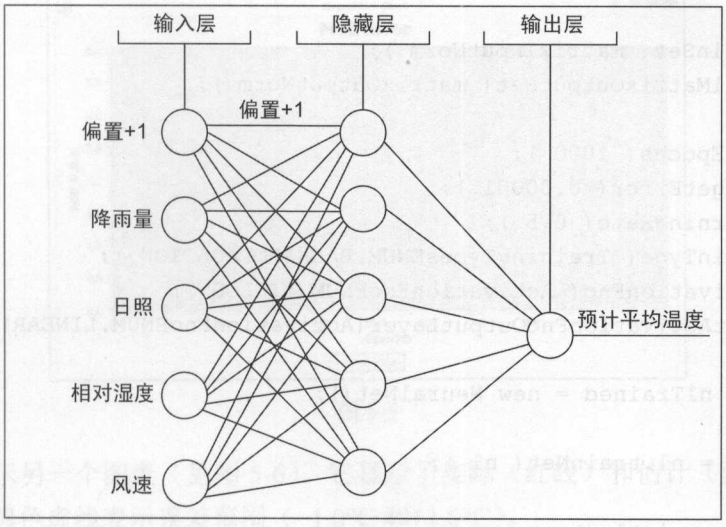


图 5-4

我们设计了一个名为 Weather 的类，专门用于天气案例。它只有一个静态主要方法，只是为了阅读天气数据文件，用这些数据创建和训练神经网络，并绘制误差以进行验证。让我们来看看在这个类中如何读取数据文件：

```
    Data weatherDataInput = new Data( "data", "inmet_13_14_input.csv" );
    Data weatherDataOutput = new Data( "data", "inmet_13_14_output.csv" );
    //sets the normalisation type
    NormalizationTypesENUM NORMALIZATION_TYPE = Data.NormalizationTypes
ENUM.MAX_MIN_EQUALIZED;

    try {
      double[][] matrixInput = weatherDataInput.rawData2Matrix( weatherDataInput );
      double[][] matrixOutput = weatherDataOutput.rawData2Matrix(weatherDataOutput );

      //normalise the data
      double[][] matrixInputNorm = weatherDataInput.normalize( matrixInput,
NORMALIZATION_TYPE );
      double[][] matrixOutputNorm = weatherDataOutput.normalize( matrixOutput,
NORMALIZATION_TYPE );
```

然后，主要方法建立一个具有 4 个隐藏层神经元的神经网络，并设置训练数据集，如下面的代码所示：

```
    NeuralNet n1 = new NeuralNet();
    n1 = n1.initNet(4, 1, 4, 1);
```

```
    n1.setTrainSet( matrixInputNorm );
    n1.setRealMatrixOutputSet( matrixOutputNorm );

    n1.setMaxEpochs( 1000 );
    n1.setTargetError( 0.00001 );
    n1.setLearningRate( 0.5 );
    n1.setTrainType( TrainingTypesENUM.BACKPROPAGATION );
    n1.setActivationFnc( ActivationFncENUM.SIGLOG );
        n1.setActivationFncOutputLayer(ActivationFncENUM.LINEAR);

    NeuralNet n1Trained = new NeuralNet();

    n1Trained = n1.trainNet( n1 );

    System.out.println();
```

这里，网络经过训练，然后，绘制误差图表。以下几行代码显示如何使用图表类：

```
    Chart c1 = new Chart();
    c1.plotXYData( n1.getListOfMSE().toArray(), "MSE Error", "Epochs", "MSE
Value" );

    //TRAINING:
    double[][] matrixOutputRNA = n1Trained.getNetOutputValues( n1Trained );
    double[][] matrixOutputRNADenorm = new Data().denormalize( matrixOutput,
matrixOutputRNA, NORMALIZATION_TYPE);

    ArrayList<double[][]> listOfArraysToJoin = new ArrayList<double[][]>();
    listOfArraysToJoin.add( matrixOutput );
    listOfArraysToJoin.add( matrixOutputRNADenorm );

    double[][] matrixOutputsJoined = new Data().joinArrays( listOfArraysToJoin );

    Chart c2 = new Chart();
    c2.plotXYData( matrixOutputsJoined, "Real x Estimated - Training Data",
"Weather Data", "Temperature (Celsius)", Chart.ChartPlotTypeENUM.COMPARISON );
```

在图 5-5 中，可以看到绘制的 MSE 训练误差。*X* 轴表示 1000 点（训练的迭代次数），并且 *Y* 轴示出 MSE 值的变化。注意到 MSE 值确立在 100 次迭代之前。

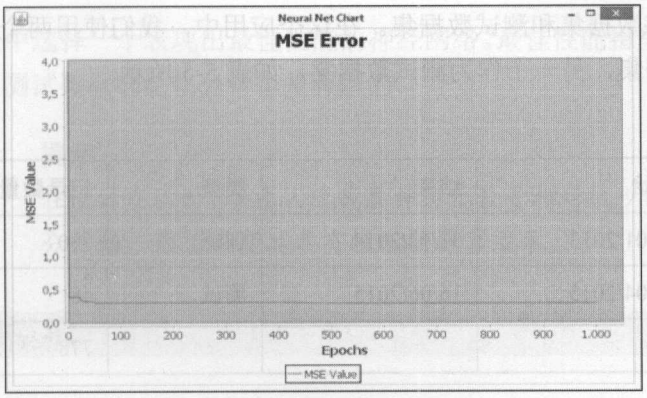


图 5-5

接下来显示另一个图表（见图 5-6）。它显示了实际（红线）和估计（蓝线）平均温度之间的比较。黑色虚线表示误差范围（−1.0°C 和+1.0°C）。

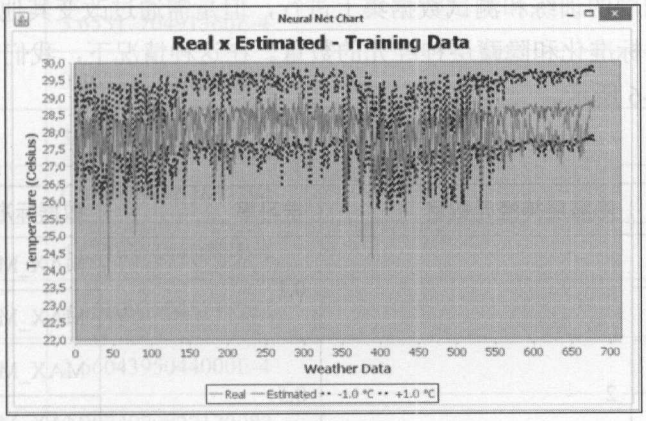


图 5-6

5.5 神经网络经验设计

当在回归问题（包括预测）中使用神经网络时，没有固定数量的隐藏层神经元。所以通常，选择任意数量的神经元，然后根据所创建的网络产生的结果改变它。该过程可以重复多次，直到找到具有满足标准的网络。

5.5.1 选择训练和测试数据集

为了证明神经网络有正确响应新数据的能力，准备两个独立的数据集是有用的，这两

个数据集称为训练数据集和测试数据集。在这个应用中，我们使用两个不同时期的数据，一个作为训练数据集，另一个作为测试数据集，如表 5-5 所示。

表 5-5

周期	开始	结束	类型	记录数量	%
1	01/01/2013	31/12/2014	训练	730	93.8
2	30/04/2015	16/06/2015	测试	48	6.2
总和				778	100

建议训练数据集至少具有总数据集的 75%。

5.5.2 设计实验

实验可以在相同的训练和测试数据集上进行，但是需通过改变其他网络参数实现，例如改变学习速率、标准化和隐藏层神经元的数量。在这种情况下，我们进行了 12 个实验，其参数选择如表 5-6 所示。

表 5-6

实验	隐藏层神经元数目	学习率	数据标准化类型
1	2	0.1	MAX_MIN
2			MAX_MIN_EQUALIZED
3		0.5	MAX_MIN
4			MAX_MIN_EQUALIZED
5		0.9	MAX_MIN
6			MAX_MIN_EQUALIZED
7	4	0.1	MAX_MIN
8			MAX_MIN_EQUALIZED
9		0.5	MAX_MIN
10			MAX_MIN_EQUALIZED
11		0.9	MAX_MIN
12			MAX_MIN_EQUALIZED

目的是从实验中选择一个表现出最佳性能的神经网络。最佳性能指呈现最低 MSE 误差的网络，但是使用测试数据的泛化分析也是有用的。

提示：

在设计实验时，考虑从隐藏层神经元数量相对较少的部分开始，因为期望其具有较低的计算成本。

5.5.3 结果和模拟

运行 12 个实验后，我们发现以下 MSE 误差，如表 5-7 所示。

表 5-7

实验	MSE 误差
1	3.6551720491360E-4
2	0.3034120360203837
3	3.8543681112765E-4
4	0.3467096464653794
5	4.6319274448088E-4
6	0.4610935945738937
7	2.6604395044000E-4
8	0.2074979827120087
9	2.7763926432754E-4
10	0.2877786584371894
11	3.4582006086257E-4
12	0.4610935945709355

图 5-7 显示了神经网络的实验 5 中实际值和估计值之间的比较，以及相应的误差范围。

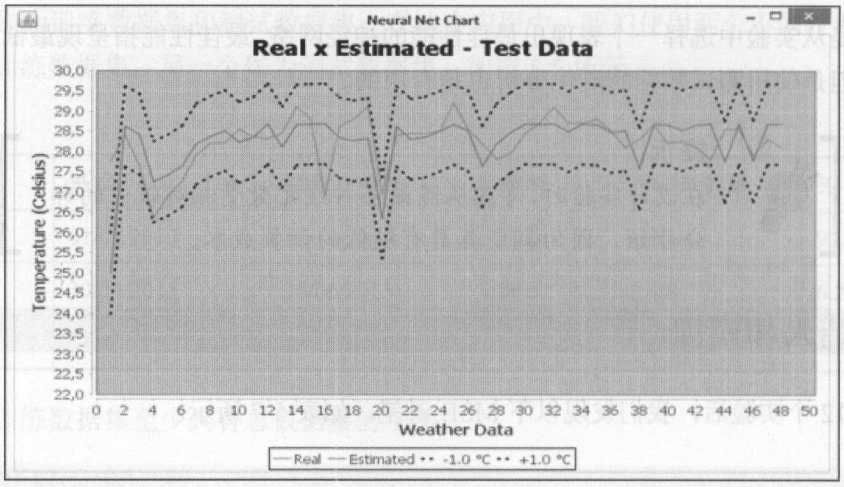


图 5-7

图 5-8 显示了与前一段所讨论的相同的结果，但对于神经网络实验 10 却并非如此。

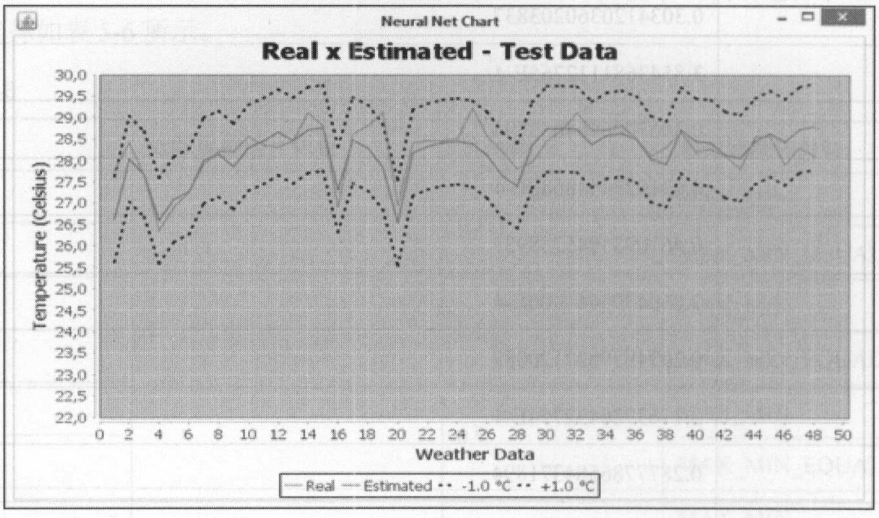


图 5-8

虽然实验 10 具有比实验 5 更大的 MSE 误差，但是实验 10 的图表可呈现更好的泛化行为。因此，我们可以得出以下结论。

- 仅考虑最终 MSE 值来决定神经网络质量是不推荐的。
- 实验 10 的估计值比实验 5 的实际值更接近实际值。

- 在实验 10 中获得的神经网络保持上升和下降的趋势比在实验 5 中获得的更好，如在天气数据 1 和 17 之间可以看到的。

因此，通过查看相应的图表，我们选择实验 10 获得的神经网络作为最适合于天气预测的神经网络。

5.6 小结

在本章中，我们已经看到了神经网络的一个有趣的实际应用。天气预报一直是一个丰富的研究领域，事实上，神经网络广泛应用于这些任务。在本章中，读者还学习了如何为预测问题准备类似的实验。用于数据选择和预处理技术的正确应用可以在设计用于预测的神经网络时节省大量的时间。本章还作为以下章节的基础，所有这些章节都将重点放在实际案例上，因此这里学到的概念将在本书的其余部分得到广泛的探讨。

在下一章中，我们将介绍分类任务，这是可以使用神经网络的另一个常见的研究领域。将提出两个案例研究，涵盖如何构建用于疾病诊断的神经网络整个过程。

第 6 章
疾病诊断分类

在本章中，读者将会看到一个非常有说服力但有趣的应用，神经网络适合于这种应用：疾病诊断。到目前为止，我们已经发现神经网络可以非常好地应用于分类问题，其中一个是想要自动分配一些记录到某个类别。本章通过介绍如何使用神经网络设计分类算法的基础知识来深入了解这一点。本章涵盖的主题如下：

- 分类问题基础知识
- 逻辑回归
  - 二分类和多分类
  - 混淆矩阵
  - 敏感性和特异性
- 分类问题神经网络
  - Java 代码改进
- 利用神经网络进行疾病诊断
  - 癌症诊断
  - 糖尿病诊断

6.1 什么是分类问题，以及如何应用神经网络

神经网络真正擅长的是分类。一个非常简单的感知机网络可以判定边界，这个边界定义数据点是属于特定区域还是属于另一区域，其中区域表示类。让我们来看看 x-y 散点图，如图 6-1 所示。

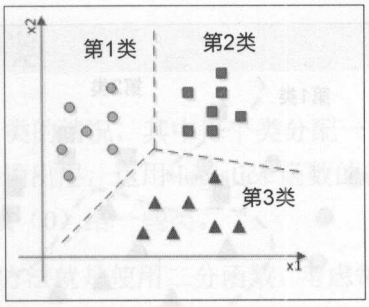


图 6-1

虚线将点明确地分成了类。这些点表示最初具有相应类标签的数据记录。这意味着它们属于哪类是已经确定，因此，这种分类任务属于监督学习。

分类算法试图寻找在多维空间中数据的类边界。一旦定义了分类边界，未分类的新数据点就可以根据分类算法定义的边界接收类标签。图 6-2 显示了新记录如何分类的示例。

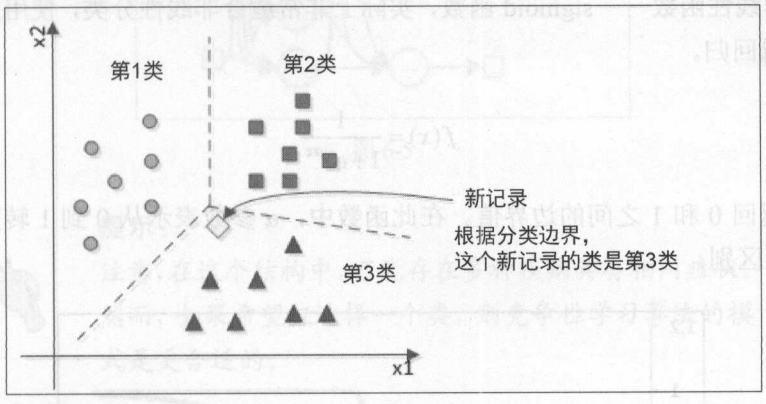


图 6-2

根据当前类的结构，新纪录是属于第 3 类的。

6.2 激活函数的特殊类型——逻辑回归

我们已经介绍了神经网络可以作为数据分类器，通过数据在多维空间中建立决策边界。该边界在感知机神经网络中是线性的，但在其他神经网络结构（例如多层感知机网络、Kohonen 神经网络或 Adaline 神经网络）的情况下是非线性的。线性情况基于线性回归，其中分类边界字面上是一条线，如图 6-2 所示。如果数据的散点图看起来像图 6-3 所示，则需要非线性分类边界。

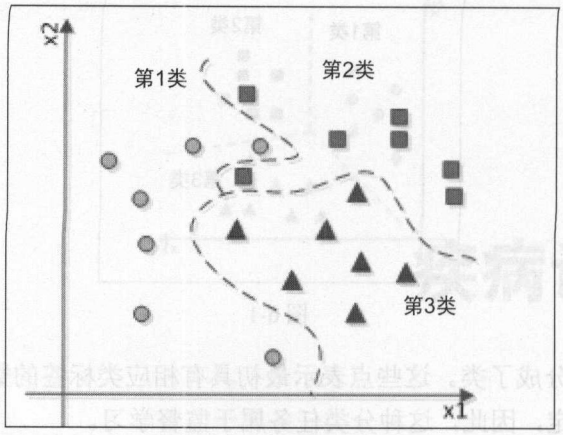


图 6-3

事实上，神经网络是一个很强大的非线性分类器，这是通过使用非线性激活函数来实现的。一个非线性函数——sigmoid 函数，实际上非常适合非线性分类，使用此函数的分类过程称为逻辑回归。

$$f(x)=\frac{1}{1+e^{-ax}}$$

该函数返回 0 和 1 之间的边界值。在此函数中，α 参数表示从 0 到 1 转变的倾斜度。图 6-4 显示了区别。

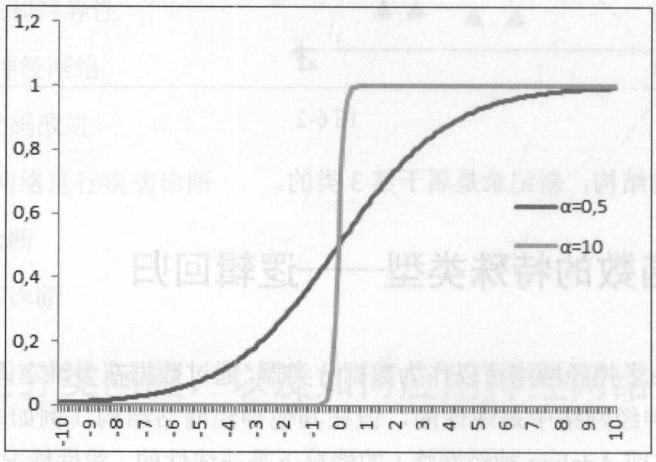


图 6-4

注意，α 参数的值越大，逻辑函数越呈现严格限制阈值函数的形状，也称为阶跃函数。

6.2.1 二分类 VS 多分类

分类问题通常是处理多个类的情况，其中每个类分配一个标签。然而，神经网络也应用二元分类模式。这是因为在输出层，运用 logistics 函数的神经网络只能产生 0 和 1 之间的值，这意味着它分配（1）或（0）给一些类。

然而，对于多分类有一种方法就是使用二分函数。考虑每个类由一个输出神经元表示，并且每当该输出神经元激活时，输入数据记录就属于该神经元对应的类。所以，让我们假设用一个网络来分类疾病，每个输出神经元表示针对某些症状的疾病，如图 6-5 所示。

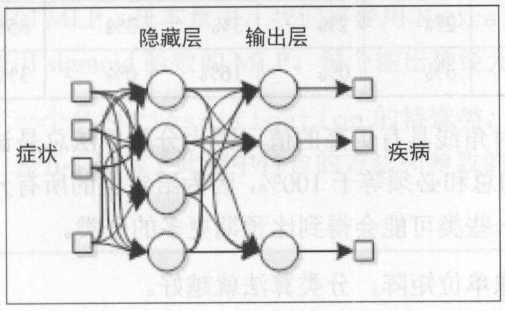


图 6-5

提示：

注意，在这个结构中，可能存在多种疾病具有相同症状。然而，如果希望仅选择一个类，则竞争性学习算法的模式是更合适的。

6.2.2 比较预期结果与产生结果——混淆矩阵

没有完美的分类器算法；所有这些算法都存在错误和偏差。然而，我们期望分类算法可以将 70%～90%的记录正确地分类。

提示：

我们并不总是期望分类算法具有非常高的分类正确率，因为输入数据可能导致分类任务出现偏差。并且，当只有训练数据被正确分类时，还可能存在过度训练的风险。

混淆矩阵表示给定类的记录中有多少被正确分类，有多少被错误分类。表 6-1 描述了混淆矩阵。

表 6-1

真实的分类	推测的分类						总和
A	B	C	D	E	F		G
A	92%	1%	0%	4%	0%	1%	2%	100%
B	0%	83%	5%	6%	2%	3%	1%	100%
C	1%	3%	85%	0%	2%	5%	4%	100%
D	0%	3%	0%	92%	2%	1%	1%	100%
E	0%	10%	2%	1%	78%	1%	8%	100%
F	22%	2%	2%	3%	3%	65%	3%	100%
G	9%	6%	0%	16%	0%	3%	66%	100%

注意，我们期望主对角线具有更高的值，因为分类算法总是试图从输入数据集中提取有意义的信息。所有行的总和必须等于 100%，因为给定类的所有元素都将分到一个可用的类中。但是，请注意，一些类可能会得到比预期更多的分类。

混淆矩阵看起来越像单位矩阵，分类算法就越好。

6.2.3 分类衡量——灵敏度和特异性

对于二分类，发现其混淆矩阵是一个简单的 2×2 矩阵，因此，其位置有特别命名，如表 6-2 所示。

表 6-2

真实的分类	推测的分类	
	真	假
阳性	真阳性	假阳性
阴性	假阴性	真阴性

疾病诊断作为本章主题，对其应用二阶混淆矩阵概念的意义在于错误诊断可以是 False Positive（假阳性，即未生病的人被诊断出疾病）或者 False Negative（假阴性，即生病的人未被诊断出疾病）。可以通过使用灵敏度和特异性指数来测量错误结果的比率。

灵敏度表示真阳性（True Positive）比率，它测量有多少正例（实际患病的记录）被正确地分类。

$$\text{Sensitivity} = \frac{\text{Number of True Positives}}{\text{Total of Actual Positive Records}}$$

特异性反过来代表真阴性率，它表示负例（实际未患病的记录）识别的比例。

$$\text{Sensitivity}=\frac{\text{Number of True Negatives}}{\text{Total of Actual Negative Records}}$$

人们期望灵敏度和特异性具有很大的值；然而，根据实际应用，灵敏度可能更具有意义。

6.3 应用神经网络进行分类

分类任务可以通过使用本书迄今为止涵盖的所有监督神经网络来执行。但是，建议使用更复杂的体系结构，例如 MLP。在本章中，我们将使用 `NeuralNet` 类来构建一个具有一个隐藏层并且输出层使用 sigmoid 函数的 MLP。每个输出神经元表示一个类。

我们在框架中添加了一个名为 `Classification` 的特殊类，以便处理诸如混淆矩阵、灵敏度和特异性等概念。表 6-3 显示了此类中包含的方法和参数的列表。

表 6-3

类名：Classfication	
方法	
`public double[][] calculateConfusionMatrix( double marginError, double[][] matrix )`	计算混淆矩阵的方法
	参数：误差范围和真实输出和预估输出矩阵
	返回：混淆矩阵
`public void printConfusionMatrix( double[][] matrix )`	输出混淆矩阵的方法
	参数：混淆矩阵
	返回：-
`public double calculateSensitivity( double[][] matrix )`	计算分类敏感度的方法
	参数：真实输出和预估输出的矩阵
	输出：敏感度值
`public double calculateSpecificity( double[][] matrix )`	计算分类特异性的方法
	参数：真实输出和预估输出的矩阵
	输出：特异性值
`public double calculateAccuracy( double[][] matrix )`	计算分类准确度的方法
	参数：真实输出和预估输出的矩阵
	输出：准确度值

续表

方法	
public double[][] convertToOneColumn(double[][] matrix)	将具有多个列的矩阵转换为一列的方法。当神经网络在输出层有多个神经元时，它会被使用
	参数：列数大于1的矩阵
	返回：只含一列的矩阵
Java 实现类：Classfication.java	

用于分类的神经网络的实现将遵循以下步骤：

1. 数据加载（训练和测试数据）；
2. 数据标准化；
3. 创建神经网络；
4. 训练神经网络；
5. 通过分类对象分析并得出分类器的结论。

首先，我们加载数据并将其标准化：

```
    //Training data
    Data dataInput = new Data("data", "inputs_training.csv");
    Data dataOutput = new Data("data", "output_training.csv");
    // test data
    Data dataInputTestRNA = new Data("data", "inputs_test.csv");
    Data dataOutputTestRNA = new Data("data", "output_test.csv");

    // normalization
    NormalizationTypesENUM NORMALIZATION_TYPE = Data.NormalizationTypesENUM.
MAX_MIN_EQUALIZED;
```

重要的是将数据转换为矩阵格式，以便可以将其传入神经网络：

```
        //convert the raw data to matrix
        double[][] matrixInput = dataInput.rawData2Matrix( diseaseDataInput );
        double[][] matrixOutput = dataOutput.rawData2Matrix( diseaseDataOutput );
        //Normalize the data. Normalization code for test data is suppressed.
        double[][] matrixInputNorm = dataInput.normalize(matrixInput,
NORMALIZATION_TYPE);
```

现在，让我们在这里创建具有 8 个输入、3 个隐藏神经元和两个输出的神经网络：

```
NeuralNet n1 = new NeuralNet();
n1 = n1.initNet(8, 1, 3, 2);
```

接下来，执行训练。因为已经在第 3 章中，看到了如何设置，所以我们略去它以节省空间。然后，我们创建一个新的网络接收训练了的网络：

```
//Create a new network to receive the trained network
NeuralNet n1Trained = new NeuralNet();
n1Trained = n1.trainNet(n1);

//Plot the error:
Chart c1 = new Chart();
c1.plotXYData(n1.getListOfMSE().toArray(), "MSE Error", "Epochs", "MSE Value");
```

训练完成后，我们实例化一个分类对象，对结果进行分析：

```
  Classification classif = new Classification();

  //Load the test data:
  n1Trained.setTrainSet( matrixInputTestRNANorm );
  n1Trained.setRealMatrixOutputSet( matrixOutputTestRNA );

  double[][] matrixOutputRNATest = n1Trained.getNetOutputValues(n1Trained);

  //Check the number of outputs to adapt the test data to the neural multiple outputs
  if(n1Trained.getOutputLayer().getNumberOfNeuronsInLayer() > 1) {

  matrixOutputTestRNA = classif.convertToOneColumn(matrixOutputTestRNA);
  matrixOutputRNATest = classif.convertToOneColumn(matrixOutputRNATest);

}
```

最后，我们应用一些处理来展示图表和混淆矩阵：

```
  ArrayList<double[][]> listOfArraysToJoinTest = new ArrayList<double[][]>();
  listOfArraysToJoinTest.add( matrixOutputTestRNA );
  listOfArraysToJoinTest.add( matrixOutputRNATest );
  double[][] matrixOutputsJoinedTest = new Data().joinArrays(listOfArrays
ToJoinTest);
```

```
        //Plot a bar chart
        Chart c3 = new Chart();
        c3.plotBarChart(matrixOutputsJoinedTest, "Real x Estimated - Test Data",
" Data", "Result (0: NO / 1: YES)");

        //plots the confusion matrix and the sensitivity and specificity indexes
        double[][] confusionMatrix = classif.calculateConfusionMatrix(0.6, matrixOutputs
JoinedTest);
        classif.printConfusionMatrix(confusionMatrix);
        System.out.println("SENSITIVITY="+classif.calculateSensitivity (confusionMatrix));
        System.out.println("SPECIFICITY="+classif.calculateSpecificity (confusionMatrix));

        //Finally the final accuracy of classification
        System.out.println("ACCURACY = " + classif.calculateAccuracy (confusion
Matrix));
```

6.4 神经网络的疾病诊断

对于疾病诊断，我们将使用免费数据集 proben1，其可在网络上获得。proben1 是来自不同领域的几个数据集的基准集。我们将使用癌症和糖尿病数据集，添加了两个新类来运行每个例子的实验：CancerDisease 和 DiabetesDisease。

6.4.1 使用神经网络诊断乳腺癌

10 个变量组成乳腺癌数据集，其中 9 个是输入，一个是二进制输出（只含 0 和 1）。数据集有 699 条记录，但我们从中排除了 16 条记录，因为发现这些记录是不完整的。因此，使用 683 条记录来训练和测试神经网络。

提示：

在实际问题中，通常有缺少或无效的数据。理想情况下，分类算法必须处理这些记录，但有时，建议排除它们，因为这些信息不会产生准确的结果。

表 6-4 显示此数据集的配置。

表 6-4

变量名称	类型	最大值和最小值
诊断结果	OUTPUT	[0; 1]
团块厚度	INPUT #1	[1; 10]
细胞大小的均匀性	INPUT #2	[1; 10]
细胞形状的均匀性	INPUT #3	[1; 10]
边缘粘连	INPUT #4	[1; 10]
单上皮细胞大小	INPUT #5	[1; 10]
裸核	INPUT #6	[1; 10]
Bland 染色质	INPUT #7	[1; 10]
正常核仁	INPUT #8	[1; 10]
有丝分裂	INPUT #9	[1; 10]

因此，所推荐的神经拓扑如图 6-6 所示。

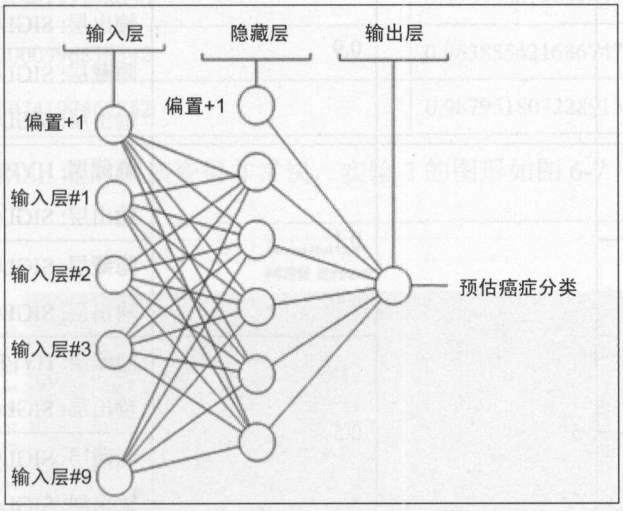


图 6-6

数据集划分如下。

- 训练：600 条记录。
- 测试：83 条记录。

与前面的例子一样，我们进行了许多实验，试图找到最好的神经网络来分类癌症是良性还是恶性。因此，我们进行了 12 个不同的实验来分析 MSE 和精度值。之后，使用测试数据集生成混淆矩阵、灵敏度和特异性并进行分析。最后，进行归纳分析。参与实验的神经网络如表 6-5 所示。

表 6-5

实现	隐藏层神经元数	学习率	激活函数
1	3	0.1	隐藏层: HYPERTAN 输出层: SIGLOG
2			隐藏层: SIGLOG 输出层: SIGLOG
3		0.5	隐藏层: HYPERTAN 输出层: SIGLOG
4			隐藏层: SIGLOG 输出层: SIGLOG
5		0.9	隐藏层: HYPERTAN 输出层: SIGLOG
6			隐藏层: SIGLOG 输出层: SIGLOG
7	5	0.1	隐藏层: HYPERTAN 输出层: SIGLOG
8			隐藏层: SIGLOG 输出层: SIGLOG
9		0.5	隐藏层: HYPERTAN 输出层: SIGLOG
10			隐藏层: SIGLOG 输出层: SIGLOG
11		0.9	隐藏层: HYPERTAN 输出层: SIGLOG
12			隐藏层: SIGLOG 输出层: SIGLOG

每次实验后，我们收集 MSE 值（如表 6-6 所示）。实验 7 和实验 12 得到最高的精度值，两个的 MSE 训练率都是可以接受的。

表 6-6

实验	MSE 训练率	准确率
1	0.03972135063712551	0.975903614457831
2	0.03995188471687546	0.975903614457831
3	0.03933513091403112	0.975903614457831
4	0.03930199248652969	0.975903614457831
5	0.04320989863852442	0.963855421686747
6	0.03906524721664331	0.975903614457831
7	0.02833532990528998	0.987951807228915
8	0.02996896005224385	0.975903614457831
9	0.02516212161358099	0.975903614457831
10	0.02510190111178650	0.975903614457831
11	0.02062000996870342	0.963855421686747
12	0.02466074197562852	0.987951807228915

从图形上看，MSE 随时间演变得非常快，实验 7 的图形如图 6-7 所示。

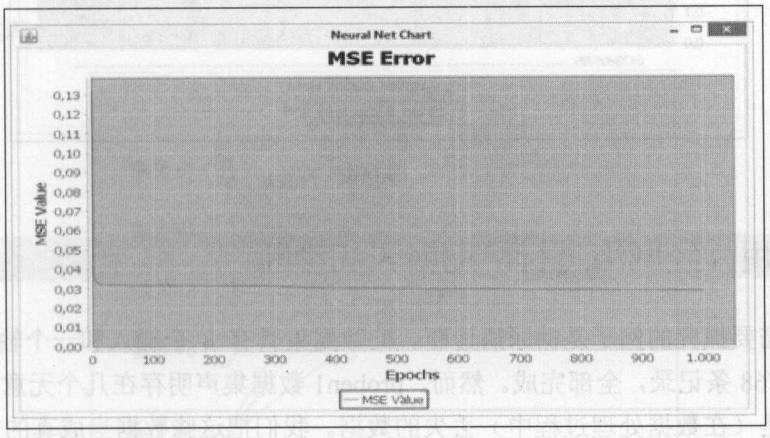


图 6-7

混淆矩阵显示在表 6-7 中，还包括两个实验的灵敏度和特异性数据。

表 6-7

实验	混淆矩阵	敏感度	特异性
7	14.0 | 1.0 0.0 | 68.0	1.0	0.9855072463768
11	13.0 | 0.0 1.0 | 69.0	0.9285714285714	1.0

现在，让我们泛化地分析。条形图显示了每种情况下期望的类以及由神经网络估计的分类，通过条形图可以更好地观察到这一特征。红色条表示实际正例诊断，而蓝色条表示神经输出值。值得注意的是，当输出为 0 时，患者被诊断患有良性癌症，当输出为 1 时，患者被诊断患有恶性癌症。使用条形图可以更好地观察到这个特征，如图 6-8 所示。

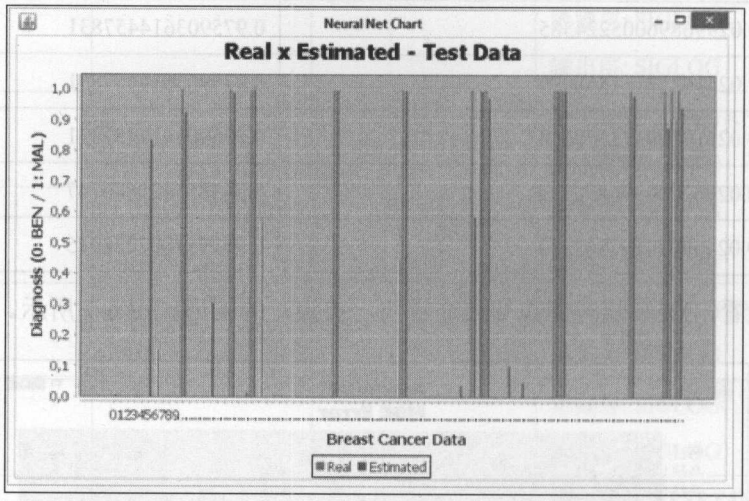


图 6-8

6.4.2 应用神经网络进行早期糖尿病诊断

另一个需要探究的例子是糖尿病诊断。此数据集具有 8 个输入和一个输出，如表 6-8 所示，共有 768 条记录，全部完成。然而，proben1 数据集声明存在几个无意义的零值，可能表示这些是（在数据处理过程中）丢失的数据。我们把这些数据当成真的来处理它们，从而引入一些错误（或噪声）到数据集。

表 6-8

变量名	类型	最大值和最小值
诊断结果	OUTPUT	[0; 1]
怀孕的次数	INPUT #1	[0.0; 17]
在口服葡萄糖耐量试验中每 2 小时的血浆葡萄糖浓度	INPUT #2	[0.0; 199]
舒张压（mm Hg）	INPUT #3	[0.0; 122]
三头肌皮褶厚度（mm）	INPUT #4	[0.0; 99]
两小时血清胰岛素（μU/ ml）	INPUT #5	[0.0; 744]
体重指数（体重，kg /（身高，m）^ 2）	INPUT #6	[0.0; 67.1]
糖尿病谱系功能	INPUT #7	[0.078; 2420]
年龄（岁）	INPUT #8	[21; 81]

数据集划分如下。

- 训练：690 条记录。
- 测试：78 条记录。

为了发现最佳的神经网络拓扑来对糖尿病进行分类，我们使用与上一节中描述的相同分析和相同的神经网络结构。然而，我们在输出层中使用多类分类：将使用该层中的两个神经元，一个用于检验存在糖尿病，另一个用于检验不存在糖尿病。

因此，推荐的神经结构如图 6-9 所示。

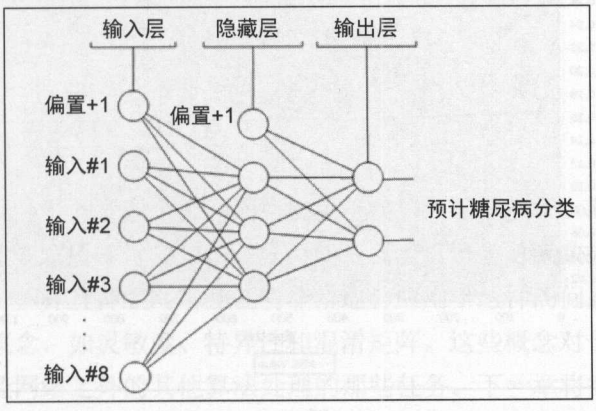


图 6-9

表 6-9 显示了前 6 个实验和最后 6 个实验的 MSE（均方误差）训练值和准确度。

表 6-9

实验	MSE 训练值	准确度
1	0.1613790087603789	0.692307692307692
2	0.1621959590254118	0.692307692307692
3	0.1643117235316208	0.653846153846153
4	0.1617892991111149	0.692307692307692
5	0.1726829994853517	0.641025641025641
6	0.1617000829026907	0.692307692307692
7	0.1568402004414977	0.666666666666666
8	0.1577266938606883	0.692307692307692
9	0.1643499270371965	0.666666666666666
10	0.1538651388477906	0.666666666666666
11	0.1747411925925356	0.692307692307692
12	0.1532305775075525	0.679487179487179

MSE 的下降速度很快，和第一个例子一样。然而，8 个实验显示了在第一次迭代的下降过程中的轻微延迟，如图 6-10 所示。

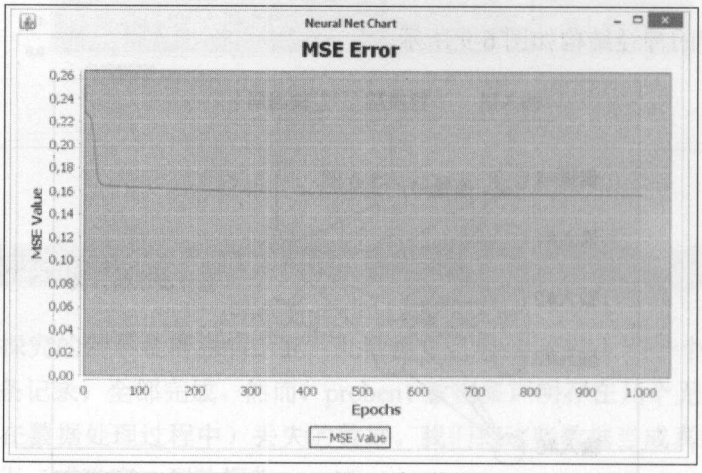


图 6-10

通过分析混淆矩阵，可以看出灵敏度和特异性没有第一个例子那样高，并且混淆矩阵的分布更均匀，如表 6-10 所示。

表 6-10

实验	混淆矩阵	敏感度	特异性
1	19.0 | 11.0 13.0 | 35.0	0.59375	0.7608695652173914
8	21.0 | 13.0 11.0 | 33.0	0.65625	0.717391304347826

虽然这可能表明分类器是无效的，因为假阳性或假阴性的数量（过多），我们应该想到原始数据集包含了不良记录，这些不良记录不能及时过滤。这也解释了为什么在泛化条形图（见图 6-11）中出现假阴性。

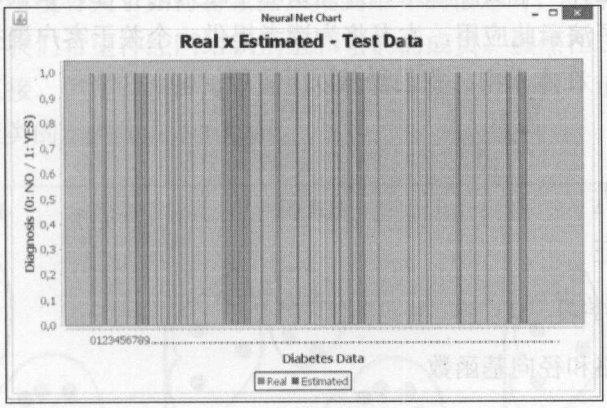


图 6-11

6.5 小结

在本章中，我们已经看到了两个神经网络应用于疾病诊断的例子。简要回顾了分类问题的基本原理，以便阐述本章所探讨的知识。分类任务属于机器学习或数据挖掘领域中最常使用的监督学习任务之一，并且神经网络经证明非常适合应用于这样的问题。还为读者介绍了用于评估分类任务的概念，如灵敏度、特异性和混淆矩阵。这些概念对于所有分类任务非常有用，包括用除了神经网络之外的其他算法处理的那些任务。下一章将探讨类似的任务，但使用无监督学习意味着没有预期的输出数据，本章中内容将对下一章的理解有所帮助。

第 7 章
客户特征聚类

无监督学习的神经网络所具有的神奇功能之一是能够找到数据中的隐藏模型，对于这些隐藏模型，甚至连专家都可能没有任何线索。在本章中，我们将通过使用事务数据库来查找客户组（簇）这样一个实际应用，并探索这个迷人的特性。我们将简单回顾无监督学习和聚类任务。为了演示此应用，本书将为读者提供一个关于客户聚类分析的实际示例及其在 Java 中的实现。在本章中，我们涵盖以下主题：

- 聚类任务
  - 聚类分析
  - 聚类验证
- 应用无监督学习
  - 神经网络和径向基函数
  - 用于聚类的 Kohonen 网络
  - 处理不同类型数据
- 客户特征分析
  - 预处理
- Java 实现
  - 信用分析和用户特征

7.1 聚类任务

聚类分析是广泛的数据分析任务中的一部分，其目的是将看起来更相似，彼此更类似的元素组合成簇或组。聚类任务完全基于无监督学习，因为不需要含有任何目标输出数据就能找到簇。相反，解决方案设计者可以选择自己想要将记录分组的数量并检查算法对其的响应。

提示：

聚类任务可能与分类任务重复，其中至关重要的区别在于聚类中，在运行聚类算法之前不需要有预定义的类集合。

当只有很少或根本没有关于如何将数据聚集到组中的信息时，人们希望应用聚类分析。假如有一个数据集，我们希望神经网络识别组和它们的成员。虽然在二维数据集中可视化地执行似乎更简单直接。但是，当维度更高时，该任务的执行变得不那么简单并且需要算法解决方案。二维聚类的示例如图 7-1 所示。

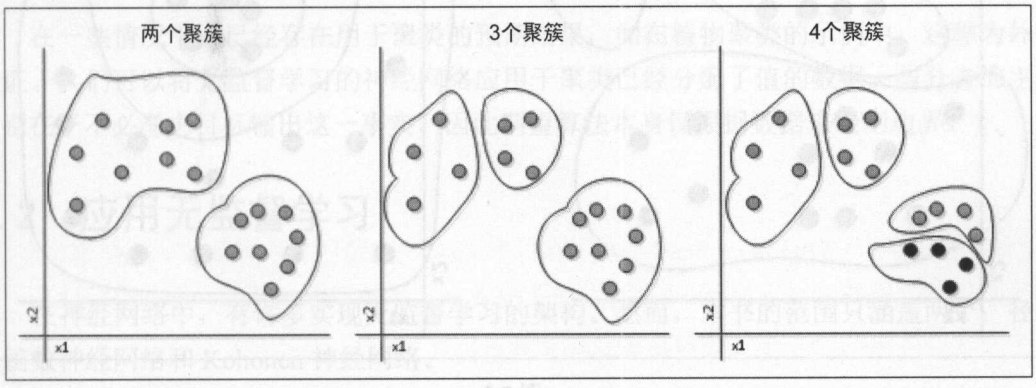


图 7-1

在聚类分析中，簇的数量不是由数据决定的，而是由期望聚集数据的数据分析人员决定的。这里，边界与分类任务的边界稍有不同，因为它们主要取决于集群的数量。

7.1.1 聚类分析

在聚类任务以及无监督学习任务中的一个难点是对结果的准确解释。在监督学习中，

有一个定义好的目标，我们可以从中导出一个误差度量或混淆矩阵；在无监督学习中，质量评估是完全不同的，完全取决于数据本身。验证标准包括评估数据在簇中的分布情况以及专家对数据的外部意见的指数，这也是对质量的度量。

提示：

例如，假设对植物进行聚类，给定其特点（尺寸、叶子颜色、结果期，等等）。如果神经网络错误地将仙人掌和松树分组在同一个簇中，植物学家根据其在该领域的专业知识肯定不会认可这样的分类，且会声明这种分类没有任何意义。

聚类中出现两个主要问题。一个是一个神经网络的输出从未被激活的事实，这意味着某个簇没有一个与其相关联的数据点；另一个是非线性或稀疏聚类的情况，其可以被错误地分组成几个簇，而实际上，可能只有一个，如图 7-2 所示。

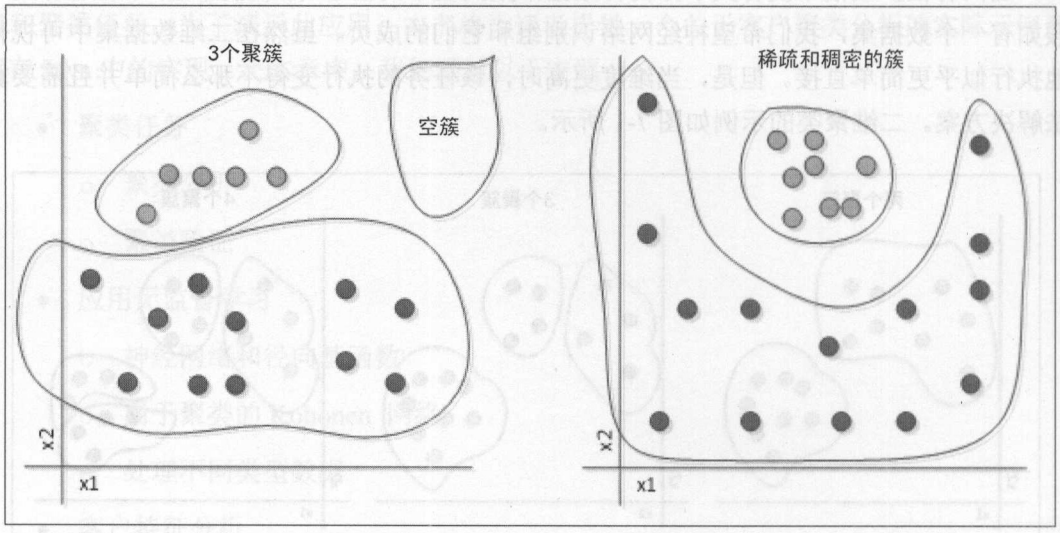


图 7-2

7.1.2 聚类评估和验证

不幸的是，如果神经网络聚集不好，需要重新定义类的数量或执行额外的数据预处理。为了评估聚类数据，可以应用 Davies-Bouldin 和 Dunn 指数。

Davies-Bouldin 指数考虑了簇的质心，以便找到簇和簇成员之间的内部距离和簇之间的距离。

$$DB = \frac{1}{n}\sum_{i=1}^{n}\max_{j\neq i}\left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}\right)$$

其中 n 是簇的数量，c_i 是簇 i 的质心，σ_i 是簇 i 中所有成员的平均距离，$d(c_i, c_j)$ 是簇 i 到簇 j 之间的距离。*DB* 指数的值越小，神经网络将越强烈地考虑将其作为簇。

然而，对于密集聚类和稀疏聚类，*DB* 指数不会给出很多有用的信息。这个限制可以用 Dunn 指数来克服：

$$D = \frac{\min_{1\leq i<j\leq n} d(i,j)}{\max_{1\leq k<n\leq n} d'(k)}$$

其中 $d(i, j)$ 是 i 和 j 之间的簇间距，并且 $d'(k)$ 是簇 k 的簇内距离。这里，Dunn 指数越高，聚类越好。因为尽管聚类可能是稀疏的，但它们仍然需要一起被分组，并且高的簇内距离将表示数据的不良分组。

7.1.3 外部验证

在一些情况下，已经存在用于聚类的预期结果，如在植物聚类的示例中，这称为外部验证。人们可以将无监督学习的神经网络应用于聚类已经分配了值的数据。与分类的主要区别在于不必考虑目标输出这一事实，因此期望算法本身仅根据数据来绘制边界。

7.2 应用无监督学习

在神经网络中，有许多实现无监督学习的架构。然而，本书的范围只涵盖两个：径向基函数神经网络和 Kohonen 神经网络。

7.2.1 径向基函数神经网络

这种神经网络结构有 3 层，并且结合了两种类型的学习，如图 7-3 所示。

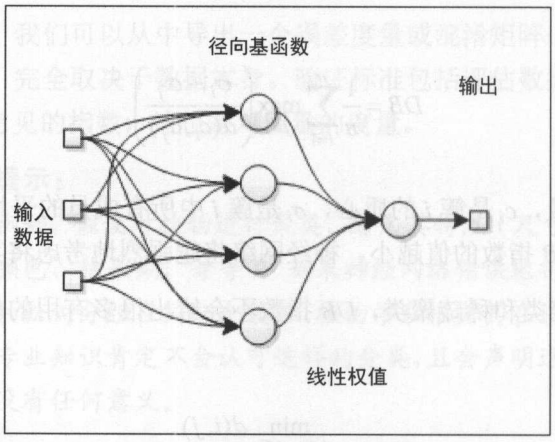


图 7-3

对于隐藏层，应用竞争学习以便激活隐藏神经元中的一个径向基函数。径向基函数采用高斯函数的形式：

$$f_i(d_i) = e^{-ad_i^2}$$

其中 d 是输入 x 和神经元 i 的权值 w 之间的距离向量：

$$d_i = \| x - w_i \|$$

神经网络的输出将是由隐藏层的神经元产生所有值的线性和：

$$y(x) = \sum_{l=1}^{N} a_i f_i(\| x - c_i \|)$$

径向基函数（radial basis functions，RBF）仅在第一隐藏层中执行聚类，而在输出层中，应用监督学习来找到输出权值。因为在 RBF 网络中定义的簇是内部的，所以我们现在不在本章中使用这个网络。第 9 章会给出详细说明。

7.2.2 Kohonen 神经网络

Kohonen 网络在第 4 章中已经涉及，现在我们主要以修改的方式来使用 Kohonen 网络。Kohonen 可以在输出端产生一维或二维的形式，但是在这里，我们只对聚类感兴趣，其可以被减少到仅一个维度。此外，簇可能彼此相关或不相关，因此在本章中可以忽略邻近神经元。这意味着只有一个神经元将被激活，并且其邻居将保持不变。因此，神经网络将调整其权值以将数据匹配簇的阵列。图 7-4 显示了 Kohonen 神经网络中的聚类层。

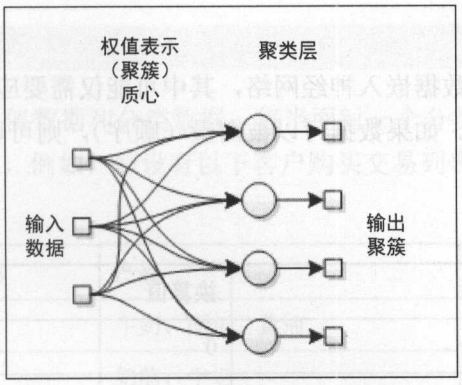


图 7-4

训练算法将使用竞争性学习算法，意味着神经网络中具有最大输出的神经元可以调整其权值。在训练结束时，期望神经网络定义了所有簇。请注意，输出神经元之间没有连接，这意味着只有一个输入在输出端有效。

7.2.3 数据类型

在实际应用中，可以通过以下方式对数据进行分类：

- 数值
  - 连续或实数
  - 离散
- 分类
  - 序数
  - 标称

提示：

到目前为止，我们主要使用数值数据，原则上，这更容易用神经网络来处理。然而，在更复杂的应用中，需要处理非数字数据，这涉及将数据转换为可应用于神经网络的“数字空间”。

数值数据的示例，例如温度（连续）和天数（离散）的值。非数值数据（分类）可以是序数，序数类型会有一个范围（例如不好、好、很好、最好），非数值数据也可以是标称类型，该类型的所有类别是同一个等级的，或者没有级别之分（例如瞳孔颜色、性别）。序数类型数据的例子是满意度（不满意、不太满意和很满意），而未定标的分类数据可以是城

市名称。

可以很容易地将数字数据嵌入神经网络，其中可能仅需要应用一些标准化或预处理。然而，分类数据需要注意。如果数据可以被衡量（顺序），则可以“离散化”。以满意度为例，我们可以创建表 7-1。

表 7-1

满意度	换算值
不满意	0
基本满意	1
非常满意	2

对于标称型数据，不建议使用那些可能引起参考变量比例缩放问题的数字。因此，最好将每个分类值视为一个二元变量，在参考值存在时表示 1，不存在时为 0，如表 7-2 所示。

表 7-2

城市名字	神经输入				
	伦敦	东京	纽约	开普敦	悉尼
London	1	0	0	0	0
Tokyo	0	1	0	0	0
New York	0	0	1	0	0
Cape Town	0	0	0	1	0
Sydney	0	0	0	0	1

这种二元变量的机制可能最终导致包含大量 0 的稀疏数据矩阵。然而，有一些技术可以解决这些问题，例如奇异值分解（single value decomposition，SVD）。读者可以在参考文献中了解更多。

7.3 客户特征

无监督学习中一个有趣的任务是客户特征分析或客户聚类分析。给定一个客户信息数据集，需要找到具有相似特征或购买相同产品的客户群。该任务为企业主创造了许多优势，这是因为该任务提供给他们关于所拥有的客户群的信息，因此能够实现更具战略性的客户关系。

数据预处理

客户信息可以包含数值数据和分类数据。每当面对一个分类标称变量，我们需要将它分成变量可能采用的数值。例如，假设有以下客户购买交易列表，如表 7-3 所示。

表 7-3

交易编号	客户编号	产品	折扣	总计
1399	56	牛奶、面包、黄油	0.00	4.30
1400	991	奶酪、牛奶	2.30	5.60
1401	406	面包、香肠	0.00	8.80
1402	239	辣椒酱、调料	0.00	6.70
1403	33	火鸡	0.00	4.50
1404	406	火鸡、黄油、调料	1.00	9.00

可以很容易地看出，产品是标称分类数据，并且对于每个交易，没有定义所购买的产品数量，即客户可以仅购买这些产品中的一个或几个单元。为了将此数据集转换为数值数据集，需要应用预处理。对于每个产品，将有一个变量添加到数据集，结果如表 7-4 所示。

表 7-4

客户编号	牛奶	面包	黄油	奶酪	香肠	辣椒酱	调料	火鸡
56	1	1	1	0	0	0	0	0
991	1	0	0	1	0	0	0	0
406	0	1	1	0	1	0	1	1
239	0	0	0	0	0	1	1	0
33	0	0	0	0	0	0	0	1

为了节省空间，我们忽略了数值变量，并认为存在客户购买此产品为 1，不存在为 0。另一个预处理方案可以考虑该值的出现次数（即该值表示顾客购买产品的次数），因此不再保持二进制值，而是变为离散。

7.4 Java 实现

在本节中，我们将基于从 Proben1（Card 数据集）收集的客户信息探索 Kohonen 神经

网络在客户聚类中的应用。

基于客户特征分析的卡信用分析

Card 数据集总共包含 16 个变量。其中 15 个是输入，一个是输出变量。出于安全原因，所有变量名已更改为无意义符号。这个数据集带来了一个很好的混合变量类型（连续的小值的分类、更大的值的分类）。表 7-5 显示了数据摘要。

表 7-5

变量	类型	值
V1	OUTPUT	−1、1
V2	INPUT #1	*b*、*a*
V3	INPUT #2	continuous
V4	INPUT #3	continuous
V5	INPUT #4	*u*、*y*、*l*、*t*
V6	INPUT #5	*g*、*p*、*gg*
V7	INPUT #6	*c*、*d*、*cc*、*i*、*j*、*k*、*m*、*r*、*q*、*w*、*x*、*e*、*aa*、*ff*
V8	INPUT #7	*v*、*h*、*bb*、*j*、*n*、*z*、*dd*、*ff*、*o*
V9	INPUT #8	continuous
V10	INPUT #9	*t*、*f*
V11	INPUT #10	*t*、*f*
V12	INPUT #11	continuous
V13	INPUT #12	*t*、*f*
V14	INPUT #13	*g*、*p*、*s*
V15	INPUT #14	continuous
V16	INPUT #15	continuous

为了简便，我们没有输入 V5-V8 和 V14，以减少过多的输入数据。此外，我们应用以下变换，如表 7-6 所示。

表 7-6

变量	类型	值	转换
V1	OUTPUT	−1、1	-
V2	INPUT #1	*b*、*a*	*b* = *1*, *a* = *0*
V3	INPUT #2	continuous	-
V4	INPUT #3	continuous	-
V9	INPUT #8	continuous	-
V10	INPUT #9	*t*、*f*	*t* = *1*,*f* = *0*
V11	INPUT #10	*t*、*f*	*t* = *1*,*f* = *0*
V12	INPUT #11	continuous	-
V13	INPUT #12	*t*、*f*	*t* = *1*,*f* = *0*
V15	INPUT #14	continuous	-
V16	INPUT #15	continuous	-

所推荐的神经网络拓扑如图 7-5 所示。

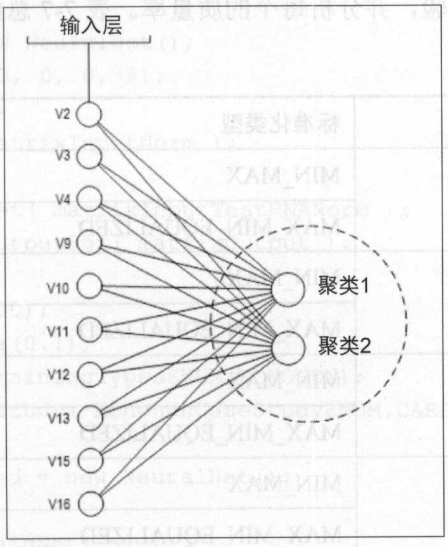


图 7-5

存储的示例数为 690，但其中 37 个具有缺失值，因此丢弃这 37 个记录。剩余的 653

个例子用于训练和测试神经网络。数据集划分如下。

- 训练：583 个记录。
- 测试：70 个记录。

Kohonen 训练算法聚类相似行为取决于一些参数，如下所示。

- 标准化类型。
- 学习率。

重要的是，注意 Kohonen 训练算法是无监督的。因此，当输出未知时可使用该算法。在 Card 示例中，在数据集中有输出值，它们在这里将仅用于证明聚类。

在这个特定的情况下，因为输出是已知的，作为分类，聚类质量可以证明如下：

- 灵敏度（真阳性率）；
- 特异性（真阴性率）；
- 准确性。

在 Java 项目中，这些值的计算是通过之前在第 6 章中开发的 `Classification` 类完成。

这是一个很好的做法，执行多次实验以试图找到最好的用户客户特征聚类的神经网络。我们将进行 10 个不同的实验，并分析每个的质量率。表 7-7 总结了将遵循的策略。

表 7-7

实验	学习率	标准化类型
1	0.1	MIN_MAX
2		MAX_MIN_EQUALIZED
3	0.3	MIN_MAX
4		MAX_MIN_EQUALIZED
5	0.5	MIN_MAX
6		MAX_MIN_EQUALIZED
7	0.7	MIN_MAX
8		MAX_MIN_EQUALIZED
9	0.9	MIN_MAX
10		MAX_MIN_EQUALIZED

Card 类创建来用于运行每个实验。关于训练，如先前在第 4 章中解释的那样，我们应用欧氏距离。

下面的代码显示了一小部分关于它的实现：

```
    Data cardDataInput = new Data("data", "card_inputs_training.csv");
    Data cardDataInputTestRNA = new Data("data", "card_inputs_test.csv");
    Data cardDataOutputTestRNA = new Data("data", "card_output_test.csv");

    NormalizationTypesENUM NORMALIZATION_TYPE = Data.NormalizationTypesENUM.
MAX_MIN;
  try {
    double[][] matrixInput = cardDataInput.rawData2Matrix( cardDataInput );

    double[][] matrixInputTestRNA = cardDataInput.rawData2Matrix(cardDataInputTestRNA);

    double[][] matrixOutput = cardDataInput.rawData2Matrix( cardDataOutputTestRNA );

    double[][] matrixInputNorm = cardDataInput.normalize(matrixInput, NORMALIZATION_
TYPE);

    double[][] matrixInputTestRNANorm = cardDataInput.normalize(matrixInputTestRNA,
NORMALIZATION_TYPE);

    NeuralNet n1 = new NeuralNet();
    n1 = n1.initNet(10, 0, 0, 2);

    n1.setTrainSet( matrixInputNorm );

    n1.setValidationSet( matrixInputTestRNANorm );
    n1.setRealMatrixOutputSet( matrixOutput );

    n1.setMaxEpochs(100);
    n1.setLearningRate(0.1);
    n1.setTrainType(TrainingTypesENUM.KOHONEN);
    n1.setKohonenCaseStudy( KohonenCaseStudyENUM.CARD );

    NeuralNet n1Trained = new NeuralNet();

    n1Trained = n1.trainNet( n1 );

    System.out.println();
    System.out.println("---------KOHONEN TEST---------");
```

```
        ArrayList<double[][]> listOfArraysToJoin = new ArrayList<double[][]>();

        double[][] matrixReal = n1Trained.getRealMatrixOutputSet();
        double[][] matrixEstimated = n1Trained.netValidation(n1Trained);

        listOfArraysToJoin.add( matrixReal );
        litOfArraysToJoin.add( matrixEstimated );
        double[][] matrixOutputsJoined = new Data().joinArrays(listOfArraysToJoin);

        //CONFUSION MATRIX
        Classification classif = new Classification();

        double[][] confusionMatrix = classif.calculateConfusionMatrix(-1.0,
matrixOutputsJoined);
        classif.printConfusionMatrix(confusionMatrix);

        //SENSITIVITY
        System.out.println("SENSITIVITY = " + classif.calculateSensitivity
(confusionMatrix));

        //SPECIFICITY
        System.out.println("SPECIFICITY = " + classif.calculateSpecificity
(confusionMatrix));

        //ACCURACY
        System.out.println("ACCURACY = " + classif.calculateAccuracy(confusionMatrix));

    } catch (IOException e) {
        e.printStackTrace();
    }
```

在使用 Card 类运行每个实验并且保存准确率之后，可以观察到实验 1 和实验 6 具有相同的精度。来自实验 1 的数据用 MIN_MAX 方法标准化，来自实验 2 的数据用 MAX_MIN_EQUALIZED 标准化，如表 7-8 所示。

表 7-8

实验	准确度
1	0.9142857142857143
2	0.6285714285714286
3	0.3714285714285714

续表

实验	准确度
4	0.6000000000000000
5	0.5857142857142857
6	0.9142857142857143
7	0.0857142857142857
8	0.3714285714285714
9	0.4142857142857143
10	0.5857142857142857

表 7-9 显示实验 1 和实验 6 的混淆矩阵、灵敏度和特异性。同样，请注意，可以观察两个实验中神经网络之间的等效性。70 个记录中只有 6 个（小于 10%）不能被正确聚类。

表 7-9

实验	混淆矩阵	敏感度	特异性
1	31.0 | 2.0 4.0 | 33.0	0.8857142857142	0.9428571428571
6	31.0 | 2.0 4.0 | 33.0	0.8857142857142	0.9428571428571

7.5 小结

在本章中，我们讨论了 Kohonen 神经网络在客户特征分析中的应用。与分类任务不同，聚类任务不考虑关于期望输出的任何前期知识。相反，我们希望神经网络发现聚类。然而，我们已经看到验证技术可能包括外部验证，这可以理解为与“目标输出”的比较。客户特征分析非常重要，因为它能为企业主提供更准确、更清晰的客户信息，而无需指出哪些客户在哪些群体中或在其他群体中的“人为干扰”，例如监督学习的情况。这是无监督学习的优势，它使数据能够单独绘制结果。

在下一章中，我们将展示神经网络的另一个有趣的应用：图像中的数字识别。这是一种在实践中了解模式识别如何与神经网络一起使用的方式。

第 8 章
模式识别（OCR 案例）

目前为止，我们已经看到了神经网络以监督方式学习和无监督方式学习来处理数据的卓越能力。在本章中，我们要呈现一个与光学字符识别（optical character recognition，OCR）相关的模式识别的额外案例。神经网络经过训练后能够准确地识别图片文件中的文本字符。在介绍应用程序本身之前，先对分类和聚类进行简要回顾。本章将涉及以下主题：

- 模式识别
  - 定义的类
  - 未定义的类
- 模式识别中的神经网络
  - Kohonen 网络和 MLP
- OCR 问题
  - 预处理和类定义
- Java 实现
  - 数字识别

8.1 什么是模式识别

模式是一些相似的数据和元素，能系统性地出现并不断重复。模式识别任务主要由无监督学习来完成。但是，当有标记数据或者已定义好分类的数据存在时，这类任务就可以用有监督方法来完成。我们作为人类完成这类任务比想像的要频繁。当我们看见物体并识别出它们属于某个确定的类别时，就是在识别一个模式。另外，当我们分析图表、离散事

件和时间序列时，或许会发现在某个特定条件下某些事件序列系统性出现的证据。总之，模式能通过数据观测值来学习。模式识别任务的案例包括，如下：

- 形状识别；
- 对象分类；
- 行为聚类；
- 语音识别；
- 光学字符识别；
- 化学反应分类。

8.1.1 定义大量数据中的类别

一个特定领域会预先定义一个类的集合，每个类都被视为一个模式。因此，每条数据或事件都会被赋予一个预先定义的类。

提示：

类通常由专家来预定义或基于应用领域的先验知识提出。另外，当我们想要将数据严格分类到一个预定义类时，最好能应用到预先定义的类。

一个使用已定义的类的模式识别的图例是动物图像识别，如图 8-1 所示。模式识别器应该被训练来捕捉正式定义类的所有特征。在本例中，显示了 8 个动物图像，属于两大类：哺乳类和鸟类。由于这是监督学习模式，所以应该提供足够数量的图片给神经网络，让它正确地对新图片进行分类。

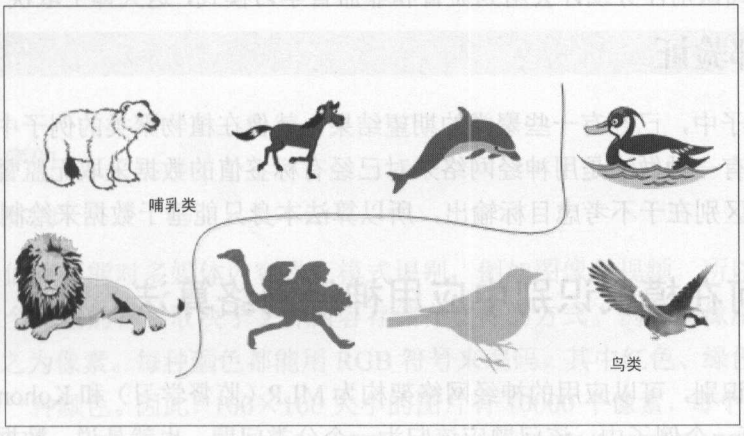


图 8-1

当然，分类可能会由于神经网络试图捕获的图片中相似的隐藏模式以及形状上细微的差别而失败。例如，虽然海豚有蹼，但它仍然是哺乳动物。有时，为了获得更好的分类效果，有必要进行数据预处理并确保神经网络收到适合的可用来分类的数据。

8.1.2 如果未定义的类没有被定义怎么办

当数据未标记并且也没有预定义类集合时，就是无监督学习的应用场景。形状识别是一个好例子，因为形状可能会比较灵活并且有无数的边、顶点或者外形，如图 8-2 所示。

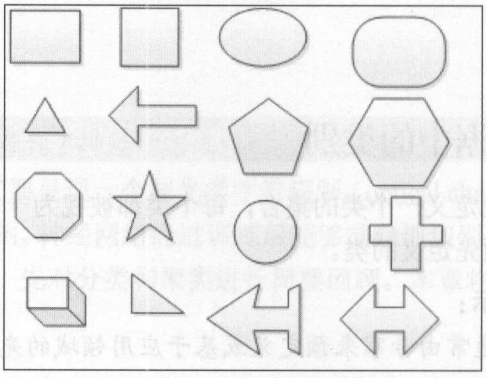


图 8-2

在图 8-2 中，能看到一些形状的类型，我们想排列它们并使得相似的形状被分组到同一个簇中。基于图像中呈现的形状信息，模式识别器可能将矩形、正方形和直角三角形分类到同一组中。但是，如果信息是作为具有边和顶点坐标的图，而不是作为图像被呈现给模式识别器，分类结果可能会有一点改变。

总之，模式识别任务或许会用到监督和非监督学习模式，这大体上取决于识别的目标。

8.1.3 外部验证

在一些例子中，已经有一些聚类的期望结果，就像在植物聚类的例子中一样，这被称为外部验证。有一种做法是用神经网络来对已经有标签值的数据采取无监督学习聚类。对于分类的主要区别在于不考虑目标输出，所以算法本身只能基于数据来绘制边界。

8.2 如何在模式识别中应用神经网络算法

对于模式识别，可以应用的神经网络架构为 MLP（监督学习）和 Kohonen 网络（非监督学习）。在第一个例子中，该问题应该归为一个分类问题，也就是说，数据应该被转换成

X-Y 数据集，X 中的每条数据记录都有一个对应的 Y 类别。如第 3 章和第 6 章所述，对于分类问题的神经网络输出都应该包含所有可能的类别，而这需要对输出记录进行预处理。

对于另外一个例子——无监督学习，就不需要对输出打上标签，但是输入数据也应该被正确的格式化。为了提醒读者，我们将这两种神经网络的模式呈现在图 8-3 中。

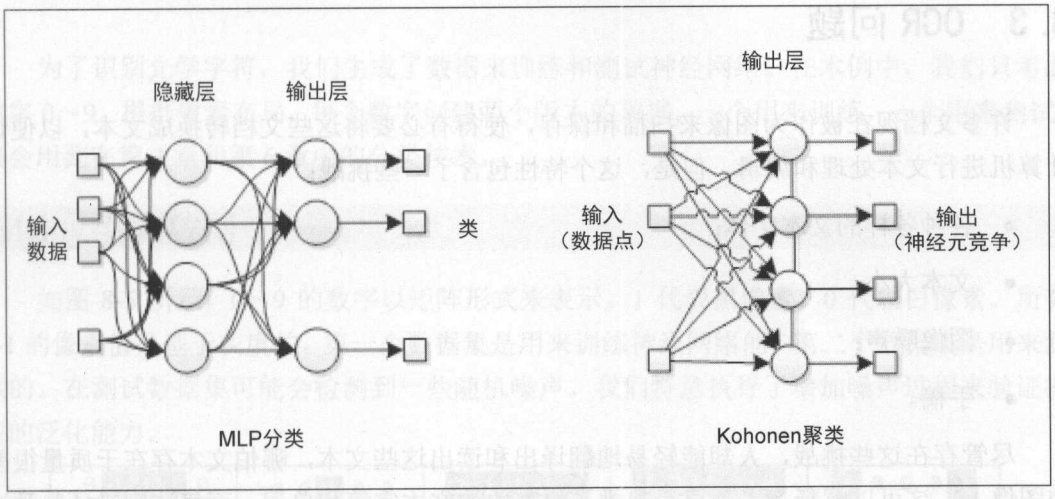


图 8-3

8.2.1 预处理数据

在模式识别里，我们必须处理所有可能的数据类型以及聚类。

- 数值型
  - 连续和真实
  - 离散
- 类别
  - 顺序的
  - 标称的

但是，我们有可能对多媒体内容进行模式识别，例如图像和视频。所以，多媒体应该如何处理？这个问题的答案取决于文件内容存储在文件的方式。例如图像用小的颜色点来表示，我们称之为像素。每种颜色都能用 RGB 符号来编码，其中红色、绿色和蓝色定义了人类可见的每一种颜色。因此，100×100 大小的图片有 10000 个像素，每个像素有 3 个值，分别代表红绿蓝，一共 30000 个点。这对于用神经网络进行图像处理是一个挑战。

一些方法，在下一章会看到，可能会将低这巨大的数据规模。那样，图像就能被作为一个连续数值的大矩阵。

为了简单起见，本章只讨论二维的灰度图像。

8.3 OCR 问题

许多文档现在被作为图像来扫描和保存，使得有必要将这些文档转换成文本，以便让计算机进行文本处理和编辑。但是，这个特性包含了一些挑战：

- 各种各样的文本字体；
- 文本大小；
- 图像噪声；
- 手稿。

尽管存在这些挑战，人却能轻易地翻译出和读出这些文本，哪怕文本存在于质量很差的图像上。这可以解释为人类已经熟悉了语言中的文本字符和单词。而算法则必须熟悉这些元素（字符、数字、信号等），以便成功地识别图像中的文本。

8.3.1 简化任务——数字识别

尽管在市场上有很多可用的 OCR 工具，但是对于一个算法来说，正确地识别图像中的文本仍然是一个很大的挑战。所以，我们会将应用限制到一个小的领域，并处理相对简单的问题。所以，在本章中，我们将会实现一个识别图像中 0～9 数字的神经网络。另外，为了简单考虑，假设图像是标准化和小尺寸的。

8.3.2 数字表示的方法

我们采用了 5×5 (25 个像素)的标准维度的灰度图像，这样每个图像都有 25 个灰度值，如图 8-4 所示。

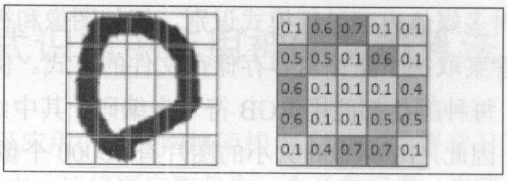


图 8-4

在图 8-4 中，左边是代表 0 的圆形，右边是相同数字的对应灰度矩阵。

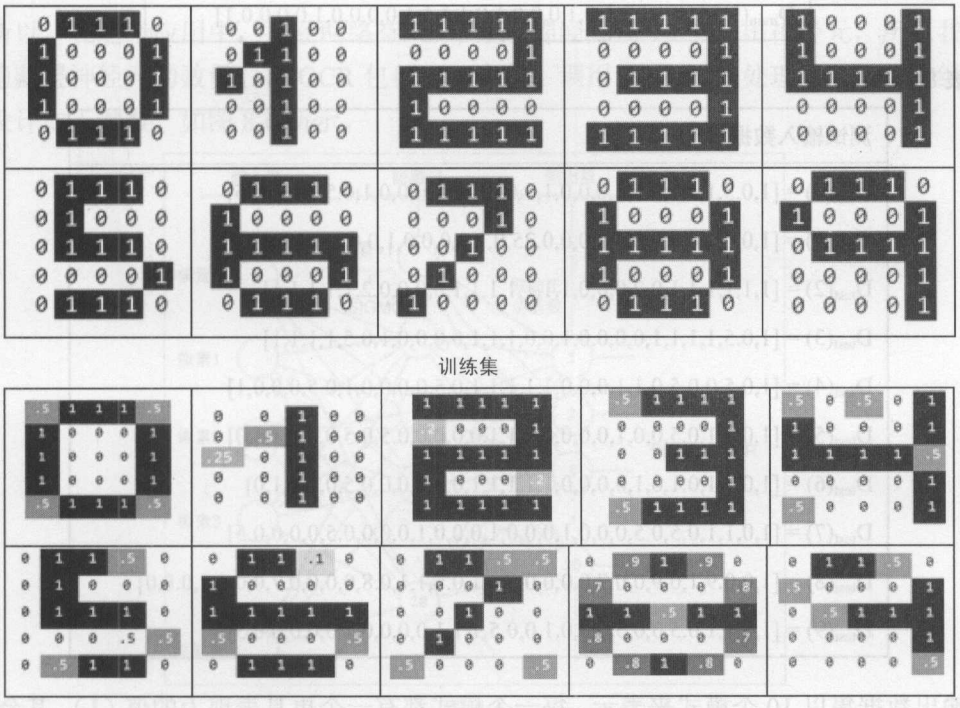
在这个应用中，我们用这个预处理过程来表示所有 10 个数字。

8.4 开始编码

为了识别光学字符，我们生成了数据来训练和测试神经网络。在本例中，我们只考虑数字 0～9。根据像素布局，每个数字创建两个版本的数据，一个用来训练，一个用来测试。将会用到在第 3 章和第 6 章中的分类技术。

8.4.1 生成数据

如图 8-5 所示，0～9 的数字以矩阵形式来表示。1 代表黑像素，0 代表白像素。所有 0-1 的像素值是基于灰度的。第一个数据集是用来训练神经网络的，第二个数据集是用来测试的。在测试数据集可能会检测到一些随机噪声。我们特意执行了增加噪声过程来验证模型的泛化能力。



训练集

测试集

图 8-5

每个矩阵行都被合并到向量（D_{train}/D_{test}）中以形成用于训练和测试神经网络的模式。所以，神经元的输入层由 26 个神经元组成。表 8-1 和表 8-2 展示了这些数据。

表 8-1

训练输入数据集
$D_{train}(0)$ = [1,0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0]
$D_{train}(1)$ = [1,0,0,1,0,0,0,1,1,0,0,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0]
$D_{train}(2)$ = [1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1]
$D_{train}(3)$ = [1,1,1,1,1,1,1,0,0,0,0,0,1,0,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1]
$D_{train}(4)$ = [1,1,0,0,0,0,1,1,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,1]
$D_{train}(5)$ = [1,0,1,1,1,1,0,0,1,0,0,0,0,0,1,1,1,1,0,0,0,0,0,1,0,0,1,1,1,1,0]
$D_{train}(6)$ = [1,0,1,1,1,1,0,0,1,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,1,0,1,1,1,0]
$D_{train}(7)$ = [1,0,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1]
$D_{train}(8)$ = [1,0,1,1,1,1,1,0,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,1,0,1,1,1,0]
$D_{train}(9)$ = [1,0,1,1,1,1,1,0,1,0,0,0,0,0,1,0,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,1]

表 8-2

测试输入数据集
$D_{test}(0)$ = [1,0.5,1,1,1,1,0.5,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,0.5,1,1,1,0.5]
$D_{test}(1)$ = [1,0,0,1,0,0,0,0,0.5,1,0,0,0,0.25,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0]
$D_{test}(2)$ = [1,1,1,1,1,1,1,0.2,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0.2,1,1,1,1,1]
$D_{test}(3)$ = [1,0.5,1,1,1,1,1,0,0,0,0,0,1,0,0,1,1,1,0,0,0,0,0,1,0,0.5,1,1,1,1]
$D_{test}(4)$ = [1,0.5,0,0,0.5,0,0,1,1,0,0,0,0,1,1,1,1,1,1,0.5,0,0,0,0,0,1,0.5,0,0,0,0,1]
$D_{test}(5)$ = [1,0,1,1,0.5,0,0,0,1,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0.5,0.5,0,0,0.5,1,1,0]
$D_{test}(6)$ = [1,0,1,1,0,0.1,0,0,1,0,0,0,0,0,0,1,1,1,1,1,1,0.5,0,0,0,0,0.5,0,1,1,1,0]
$D_{test}(7)$ = [1,0,1,1,0.5,0.5,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0.5,0,0,0,0,0.5]
$D_{test}(8)$ = [1,0,0.9,1,0.9,0,0.7,0,0,0,0,0.8,1,1,0.5,1,1,0.8,0,0,0,0,0.7,0,0.8,1,0.8,0]
$D_{test}(9)$ = [1,0,1,1,0.5,0,0.5,0,0,0,1,0,0.5,1,1,1,0,0,0,0,0,1,0,0,0,0,0.5]

输出数据集以 10 个模式来表示。每一个模式都有一个更具表现力的值（1），其余为 0。因此，神经网络的输出层将有 10 个神经元，如表 8-3 所示。

表 8-3

输出数据集
Out(0) = [0,0,0,0,0,0,0,0,0,1]
Out(1) = [1,0,0,0,0,0,0,0,0,0]
Out(2) = [0,1,0,0,0,0,0,0,0,0]
Out(3) = [0,0,1,0,0,0,0,0,0,0]
Out(4) = [0,0,0,1,0,0,0,0,0,0]
Out(5) = [0,0,0,0,1,0,0,0,0,0]
Out(6) = [0,0,0,0,0,1,0,0,0,0]
Out(7) = [0,0,0,0,0,0,1,0,0,0]
Out(8) = [0,0,0,0,0,0,0,1,0,0]
Out(9) = [0,0,0,0,0,0,0,0,1,0]

8.4.2 构建神经网络

所以，在这个应用中，神经网络有 25 个输入神经元和 10 个输出神经元，所以我们改变了隐藏层神经元的数量。在 OCR 包创建一个类：调用 `Digit` 来处理该应用。神经网络架构设计如下参数，如图 8-6 所示。

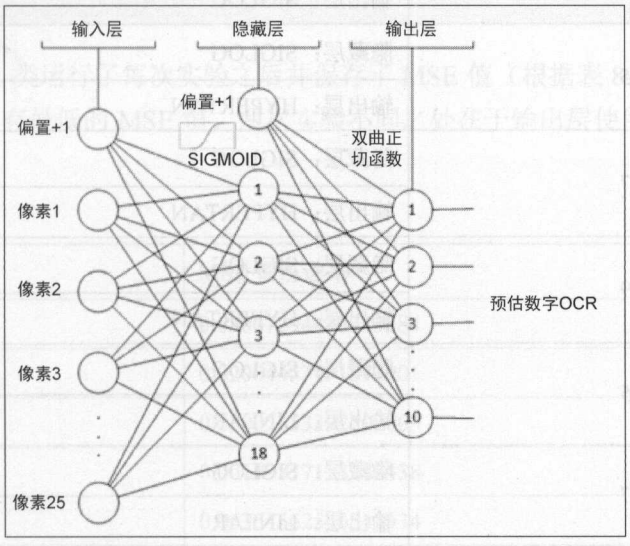


图 8-6

- 神经网络类型：MLP。

- 训练算法：反向传播。
- 隐藏层数量：1。
- 隐藏层神经元数量：18。
- 迭代次数：6000。

8.4.3 测试和重新设计——试错

现在，正如在前面提出的其他案例研究中所做的那样，让我们找到最优的神经网络拓扑结构训练出的几个网络。训练策略总结如表 8-4 所示。

表 8-4

实验	学习率	激活函数
1	0.5	隐藏层：SIGLOG
		输出层：SIGLOG
2	0.7	隐藏层：SIGLOG
		输出层：SIGLOG
3	0.9	隐藏层：SIGLOG
		输出层：SIGLOG
4	0.5	隐藏层：SIGLOG
		输出层：HYPERTAN
5	0.7	隐藏层：SIGLOG
		输出层：HYPERTAN
6	0.9	隐藏层：SIGLOG
		输出层：HYPERTAN
7	0.5	隐藏层：SIGLOG
		输出层：LINEAR
8	0.7	隐藏层：SIGLOG
		输出层：LINEAR
9	0.9	隐藏层：SIGLOG
		输出层：LINEAR

下面的 Digit 类的代码段定义了如何创建一个神经网络来读取数字数据：

```
Data ocrDataInput = new Data("data\\ocr", "ocr_traning_inputs.csv");
Data ocrDataOutput = new Data("data\\ocr", "ocr_traning_outputs.csv");
//read the data points coded in a csv file
Data ocrDataInputTestRNA = new Data("data\\ocr", "ocr_test_inputs.csv");
Data ocrDataOutputTestRNA = new Data("data\\ocr", "ocr_test_outputs.csv");

// convert these files into matrices
double[][] matrixInput = ocrDataInput.rawData2Matrix( ocrDataInput );
double[][] matrixOutput = ocrDataOutput.rawData2Matrix( ocrDataOutput );

//creates a neural network
NeuralNet n1 = new NeuralNet();
//25 inputs, 1 hidden layer, 18 hidden neurons and 10 outputs
    n1 = n1.initNet(25, 1, 18, 10);

    n1.setTrainSet( matrixInput );
    n1.setRealMatrixOutputSet( matrixOutput );

//set the training parameters
    n1.setMaxEpochs(6000);
    n1.setTargetError(0.00001);
    n1.setLearningRate( 0.7 );
    n1.setTrainType(TrainingTypesENUM.BACKPROPAGATION);
    n1.setActivationFnc(ActivationFncENUM.SIGLOG);
    n1.setActivationFncOutputLayer(ActivationFncENUM.SIGLOG);
```

8.4.4 结果

在使用 Digit 类运行了每次实验之后并保存了 MSE 值（根据表 8-5）后，我们能观察到实验 2 和实验 4 有最低的 MSE 值。两次实验不同之处在于输出层使用的学习率和激活函数，如表 8-5 所示。

表 8-5

实验	MSE 训练速率
1	0.03007294436333284
2	0.02004457991277001
3	0.03002653392502009
4	0.00119817123282438
5	0.06351562546547934
6	0.23755154264016012
7	0.19155179860965179

续表

实验	MSE 训练速率
8	1.73485602025775039
9	44.1822391373913359

MSE 随着训练迭代逐渐进化，有趣的是实验 2 的曲线在第 750 次迭代时趋于稳定，如图 8-7 所示。

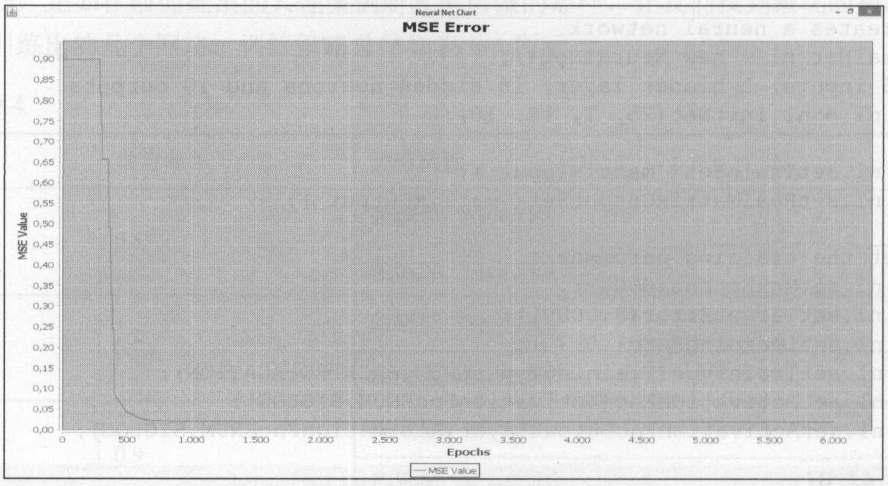


图 8-7

但是，实验 4 的曲线直到第 6000 次迭代时才保持不变，如图 8-8 所示。

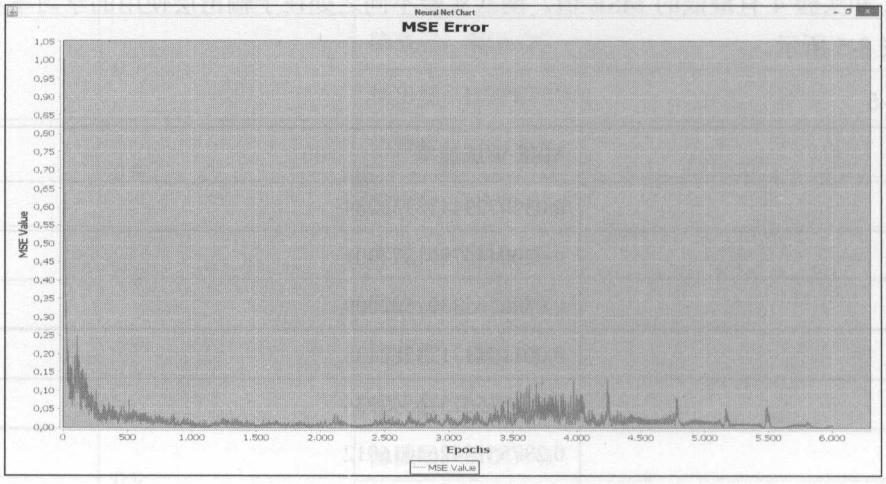


图 8-8

我们已经解释了只用 MSE 值不能完全代表神经网络的质量。相应地，测试数据集会被用来证明神经网络的泛化能力。含有噪声的实际输出和实验 2 和实验 4 的神经网络估计输出如表 8-6 所示。可以得出结论，通过实验 4 得到神经网络权值能更好地识别 0～9 的数字，即使图像呈现的像素比实验 2 得到的图像有更多的噪声。虽然实验 2 错误地分类了 3 种模式，但实验 4 正确地分类了所有模式。

表 8-6

输出对比										
实际输出值（测试数据集）										数值
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0
1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1
0.00	**1.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2
0.00	0.00	**1.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3
0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	0.00	0.00	0.00	4
0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	0.00	0.00	5
0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	0.00	6
0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	7
0.00	0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	0.00	8
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00	9
预估输出值（测试数据集）——实验 2										数值
0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.00	0.00	**0.97**	0 (OK)
0.97	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	1 (OK)
0.00	0.00	0.00	**0.02**	0.00	0.00	0.00	0.01	0.00	0.00	4 (ERR)
0.00	0.00	0.00	0.02	0.00	0.00	**0.20**	0.00	0.00	0.00	7 (ERR)
0.00	0.00	0.00	**0.96**	0.00	0.00	0.00	0.02	0.00	0.00	4 (OK)
0.01	0.00	0.00	0.00	**0.98**	0.01	0.00	0.00	0.00	0.00	5 (OK)
0.01	0.00	0.00	0.00	0.00	**0.56**	0.00	0.07	0.00	0.00	6 (OK)
0.00	0.00	0.00	0.00	**0.66**	0.00	0.14	0.00	0.00	0.00	5 (ERR)
0.00	0.00	0.00	0.00	0.00	0.03	0.00	**0.93**	0.00	0.01	8 (OK)
0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	**0.96**	0.00	9 (OK)

预估输出值（测试数据集）——实验4										数值
0.00	0.16	0.09	0.06	0.06	0.01	0.11	-0.27	-0.09	**0.97**	0 (OK)
1.00	0.00	0.09	0.13	0.21	-0.22	0.42	0.19	0.34	0.14	1 (OK)
0.00	**0.99**	0.04	0.05	0.07	0.10	0.14	0.18	0.22	0.25	2 (OK)
0.01	0.03	**0.81**	0.06	0.09	0.03	0.74	-0.03	-0.03	-0.12	3 (OK)
0.02	-0.11	-0.10	**0.94**	0.08	0.08	0.11	0.85	0.09	0.06	4 (OK)
0.02	-0.01	0.10	0.06	**1.00**	0.11	0.10	0.11	0.10	0.06	5 (OK)
-0.00	-0.07	-0.05	0.22	0.09	**1.00**	0.20	0.11	0.26	0.20	6 (OK)
0.51	-0.05	0.25	0.09	0.96	0.22	**0.99**	0.25	0.34	0.34	7 (OK)
0.00	0.04	0.04	0.04	0.05	0.06	0.05	**0.98**	0.03	0.07	8 (OK)
0.00	0.01	0.05	0.01	0.02	0.00	0.04	0.03	**1.00**	0.02	9 (OK)

8.5 小结

在本章中，我们已经感受到了神经网络关于识别图像中0～9数字的能力。尽管数字编码在5×5图像中非常小，但是它对于在实践中理解概念非常重要。神经网络有能力从数据中进行学习，但条件是真实世界的表现形式可以转换为数据。字符识别是模式识别应用的一个非常好的例子是有道理的，这里的应用可以扩展到任意类型的字符，前提是神经网络能表现所有的预定义字符。

下一章将探讨本书中的所有内容，以便为读者提供一些用于优化和改进神经网络应用的选项，并为本书做一个总结。

第 9 章
神经网络优化与自适应

在本章中，读者将会看到有助于优化神经网络的技术，从而使其达到最佳性能。例如输入选择、数据集分离和过滤等任务以及隐藏层神经元的数量选择的任务都是可以被调整以提高神经网络性能的例子。另外，本章还关注让神经网络适应实时数据的方法。本章会展示这些技术的两个实现，并选择一些应用问题来做练习。本章涉及以下主题：

- 输入选择
  - 降维
  - 数据过滤
- 结构选择
  - 剪枝
- 在线再训练
  - 随机在线学习
- 自适应神经网络
  - 自适应共振理论

9.1 神经网络实现中的常见问题

当开发一个神经网络应用时，经常会面对关于结果准确性的问题。这些问题的源头是多样的：

- 不良的输入数据；

- 噪声数据；
- 非常大的数据集；
- 不合适的结构；
- 不合适的隐藏层神经元；
- 不合适的学习率；
- 不良的数据分段。

神经网络应用的设计有时需要很多耐心和试错方法。不存在一种方法论可以特别指出隐藏单元的数量应该是多少或者应该用哪种架构，但是对于如何正确选择这些参数还是有一些推荐项的。程序员可能会面对的另一个问题是训练时间过长，这经常引起神经网络无法学习数据。无论训练过程运行了多长时间，神经网络都不会收敛。

提示：
设计一个神经网络需要程序员或者设计者按需多次测试和重新设计结构，直到得到一个可以接受的结果。

另一方面，人们希望提高结果质量。直到学习算法达到停止条件，即迭代次数或者均方误差（译者注，指达到这两个条件的预设值），神经网络才停止学习过程。即便这样，有时结果还是不够准确或者不够泛化，那就需要对神经网络结构和数据集进行重新设计。

9.2 输入选择

设计神经网络应用的关键任务之一是选择合适的输入。对于无监督学习的例子来说，人们希望只使用与结果有关的变量，这样神经网络将在其基础上找到某种模式。对于监督学习的例子来说，需要将输出和输入映射起来，所以人们需要选择能适当影响输出的输入变量。

9.2.1 数据相关性

在监督学习的例子中有助于选择优质输入的一个策略是数据序列之间的相关性。数据之间的相关性是度量一个数据序列如何作用或者影响其他序列的方式。假设我们有一个数据集包含了若干数据序列，从中选择一个作为输出。现在，我们需要从剩下的变量中选择输入。

然后，我们会一次评估一个变量对输出的影响，以便决定是否将其包括为输入。相关系数是最常用的变量之一：

$$r_{x(k)y(k)} = \frac{S_{x(k)y(k)}}{\sqrt{S_{x(k)x(k)}S_{y(k)y(k)}}}$$

其中 $S_{x(k)y(k)}$ 表示 x 和 y 变量的协方差：

$$S_{x(k)y(k)} = \sum_{i=\tau}^{n} x(i)y(i) - \frac{\sum_{j=0}^{n} x(j)\sum_{j=0}^{n} y(j)}{n}$$

相关系数的取值范围−1～1，当值接近于 1 就说明正相关，值接近于−1 就说明负相关，并且值接近于 0 说明完全没有相关性。

举例说明，下面关于两个变量 X 和 Y 的 3 个图表，如图 9-1 所示。

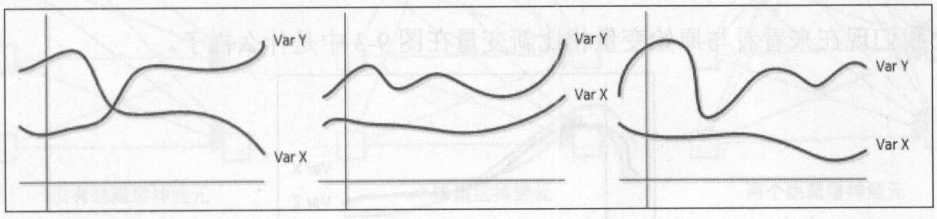


图 9-1

在左边的第一个图表里，很显然地能够发现一个变量下降而另一个变量则上升（相关系数：−0.8）。中间的图表显示了两个变量以相同的趋势在变化，所以它们的相关系数为正（相关系数：+0.7）。在右边的第三个图，显示了变量之间没有相关性（相关系数：−0.1）。

没有关于限制使用哪个相关系数这样的阈值规则，它取决于应用。相关系数绝对值大于 0.5 可能适合某个应用，而其他情况，绝对值在 0.2 左右也是一个显著的指标。

9.2.2 降维

另一个有趣的点是减少冗余数据。有时，当在无监督学习和监督学习中存在大量可用数据时，这就需要减少冗余数据了。举例说明，来看看图 9-2 中的两个变量。

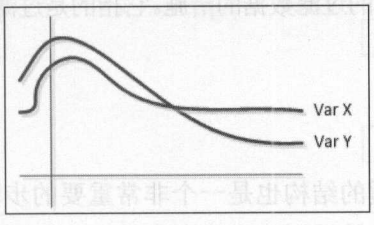


图 9-2

可以看到 X 和 Y 两个变量形状相同，所以这就可以理解为冗余，这是由于两个变量因为高度相关都携带了几乎相同的信息。因此，人们就会想到一种称为主成分分析（principal component analysis，PCA）的技术，它是处理这类例子的好方法。

主成分分析的结果将是总结了前面两个（或者更多）变量的新变量。基本上，原始数据序列会减去平均值并乘以协方差矩阵的特征向量的转置：

$$S=\begin{bmatrix} S_{XX} & S_{XY} \\ S_{YX} & S_{YY} \end{bmatrix}$$

其中 S_{XY} 代表了变量 X 和 Y 的协方差。

计算出来的新数据如下：

$$Z-eig(S)^T[X-E[X]\ \ Y-E[Y]]$$

让我们现在来看看与原始变量相比新变量在图 9-3 中是什么样子。

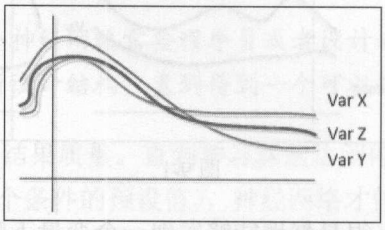


图 9-3

9.2.3 数据过滤

噪声数据和糟糕的数据也是神经网络应用的问题源头，这也是我们需要过滤数据的原因。通用数据过滤技术是指排除掉那些超过通常范围的记录。例如，温度值通常在−40～40之间，所以温度值为 50 就会被认为是异常值并被移除。

$$d_i=\left|\frac{X_i-E[X]}{std(X)}\right|\leqslant 3$$

3 sigma 原则是良好且有效的过滤数据的措施。它指的是过滤掉超过均值 3 倍标准差的数据。

9.3 结构选择

为神经网络选择一个合适的结构也是一个非常重要的步骤。但是，这通常根据经验来完成，这是因为没有关于一个神经网络应该有多少个隐藏层单元的规则。唯一关于多少个

单元更合适的度量当时就是神经网络的性能。然而，这样可能会存在一个能产生相同结果的更小一些的结构。

关于这点，大体上有两种方法论：建造型和剪枝型。建造型方法指的是一开始只确定输入层和输出层，接着开始往隐藏层增加新的神经元，直到得到一个好的结果。而破坏型方法，也称为剪枝型，它基于较大结构工作，在之上取出对输出贡献较小的神经元。

建造型方法如图 9-4 所示。

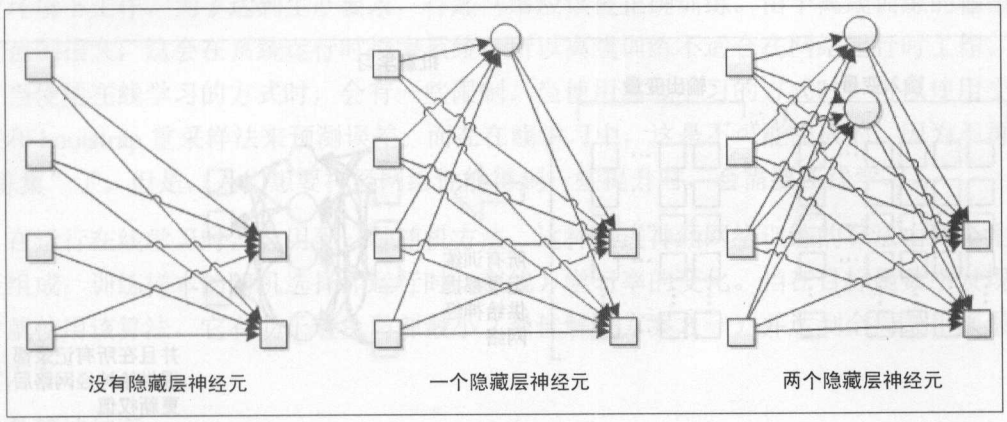


图 9-4

剪枝型方法反其道而行之。当神经元数量有很多时，“剪掉”那些敏感度很低的神经元，这意味着它对于误差的贡献最小，如图 9-5 所示。

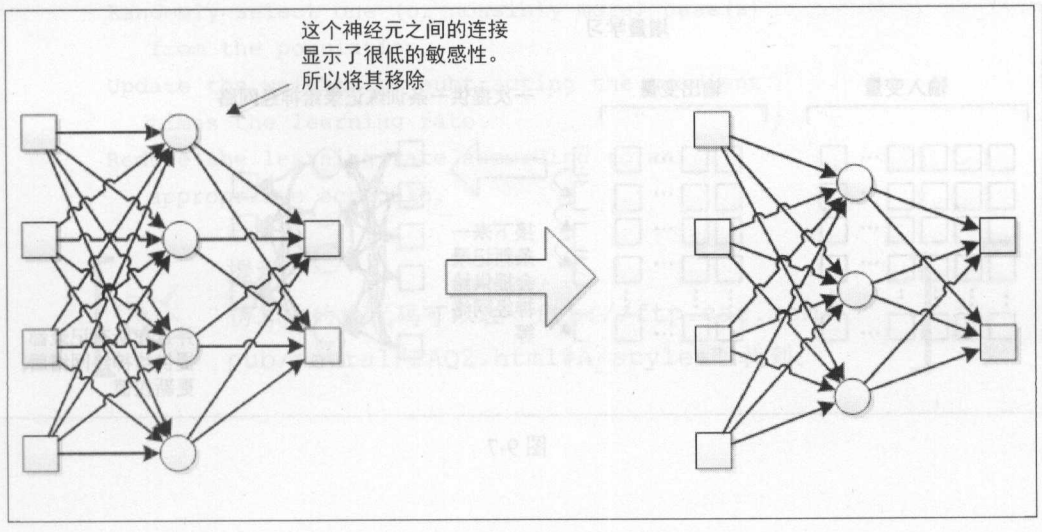


图 9-5

9.4 在线再训练

在训练过程期间，重要的是设计如何执行训练。两个基本方法是批量学习和增量学习。

在批量学习中，所以记录都被提供给神经网络，这样神经网络就能评估误差并更新权值，如图 9-6 所示。

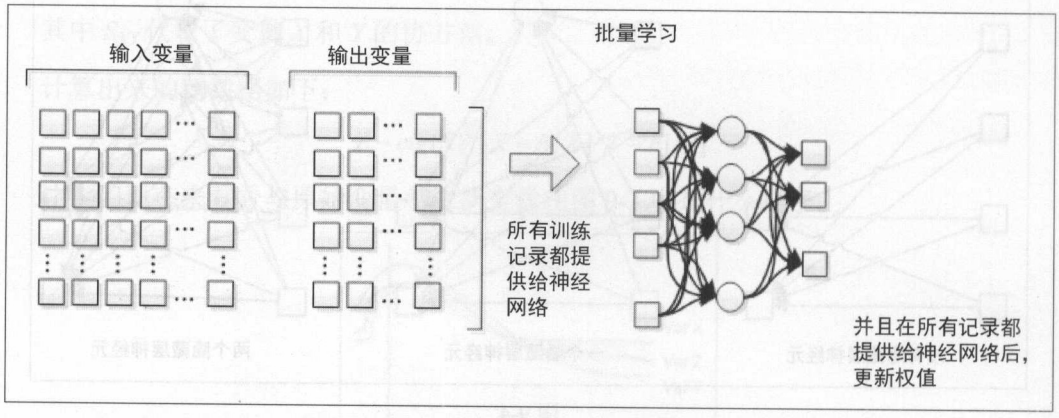


图 9-6

在增量学习中，更新操作在每条记录发送给神经网络之后执行，如图 9-7 所示。

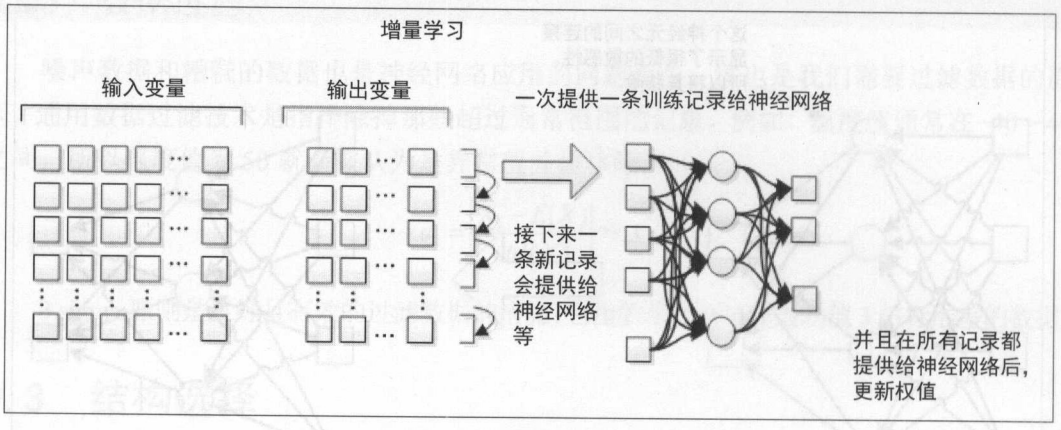


图 9-7

两种方式都很好，并各有其利弊。批量学习可以用于虽然较不频繁但更具方向性的权值更新，而增量学习提供了一种对权值精雕细琢的调整方式。在这种情况下，可以设计一种能使神经网络持续学习的模式。

9.4.1 随机在线学习

离线学习意味着神经网络在没有“运行”时学习。每一个神经网络应用都假设在某个特定环境下工作，为了达到生产要求，神经网络应该被正确训练。由于离线训练的输出值变化范围很大，这会在系统运行时损害系统，所以离线训练不适合在网络运行时工作。但是，当使用在线学习的方式时，会有一些限制。当使用离线学习的方式时，可以使用交叉验证和 bootstrap 重采样法来预测误差。而在在线学习中，这是不可能完成的，因为不再有“训练集”了。但是，人们想要神经网络性能得到一些提升时，会需要在线学习。

在运行在线学习时，会用到一种随机方法。这种改进神经网络训练的算法由两个主要特征组成：训练样本的随机选择和运行时（在线）学习率的变化。当在目标函数中发现噪声时就使用该算法。它有助于避免局部最小（最佳解决方案之一）并达到全局最小（最佳解决方案）。

伪算法如下：

```
Initialize the weights.
   Initialize the learning rate.
   Repeat the following steps:
      Randomly select one (or possibly more) case(s)
         from the population.
      Update the weights by subtracting the gradient
         times the learning rate.
      Reduce the learning rate according to an
         appropriate schedule.
```

提示：

伪算法的源代码可以在“ftp://ftp.sas.com/pub/neural/FAQ2.html#A_styles”找到。

9.4.2 实现

在 Java 项目中，已经在 learn 包中创建了 BackpropagationOnline 类。该算法和经典反向传播算法的不同之处在于 train() 方法有变化，并增加了两个方法——generateIndexRandomList() 和 reduceLearningRate()。第一个方法生成了一个索引的随机集合，在训练步骤中会用到，第二个方法根据以下启发式算法来执行学习率的在线变化：

```
private double reduceLearningRate(NeuralNet n, double percentage) {
    double newLearningRate = n.getLearningRate() *
                    ((100.0 - percentage) / 100.0);

    if(newLearningRate < 0.1) {
      newLearningRate = 1.0;
    }

    return newLearningRate;
  }
```

train() 方法也根据前面介绍的伪算法进行了修改，该方法主要部分的代码如下：

```
ArrayList<Integer> indexRandomList = generateIndexRandomList(rows);

while(getMse() > n.getTargetError()) {

  if ( epoch >= n.getMaxEpochs() ) break;

    double sumErrors = 0.0;

    for (int rows_i = 0; rows_i < rows; rows_i++) {

      n = forward( n, indexRandomList.get(rows_i) );

      n = backpropagation( n, indexRandomList.get(rows_i) );

      sumErrors = sumErrors + n.getErrorMean();

      n.setLearningRate( reduceLearningRate( n, n.getLearningRatePercentage
Reduce() ) );
    }

    setMse( sumErrors / rows );
```

```
n.getListOfMSE().add( getMse() );

epoch++;
}
```

9.4.3 应用

我们使用了前面章节的数据来测试这种训练神经网络的新方法。本章使用与第 5 章和第 8 章定义相同的神经网络拓扑。第一个是天气预测问题，第二个是 OCR。表 9-1 显示了结果对比。

表 9-1

值	天气预测	OCR
经典反向传播学习率	0.5	0.5
经典反向传播 MSE 值	0.2877786584	0.0011981712
在线反向传播学习	得到：≅ ≅ 0.15	得到：≅ ≅ 0.40
在线反向传播 MSE 值	0.4618623052	9.977909980E-6

图 9-8 显示了在使用新训练方法之后发现的 MSE 演变，它考虑了天气预测数据。曲线有一个锯齿形状是因为学习率的变化。另外，这与第 5 章中展示的曲线很相似。

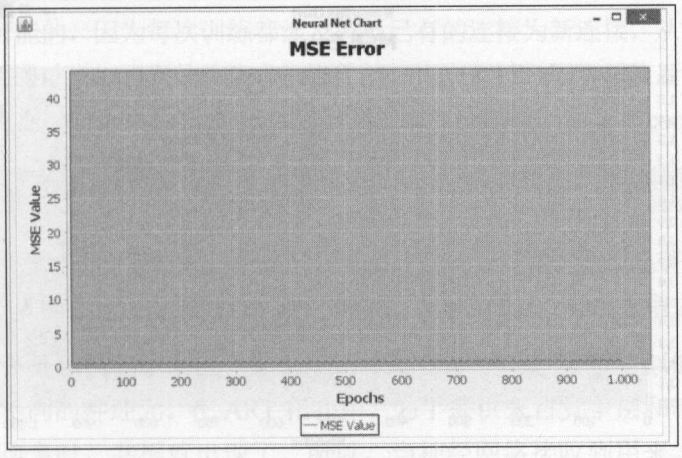


图 9-8

另一方面，图 9-9 所示是用 OCR 数据生成的，这时训练过程变得更快了并且在第 900

次迭代就停止了，这是因为它的 MSE 值已经很小了。重要的是，在第 8 章中，训练过程更加缓慢并持续到了第 6000 次迭代。

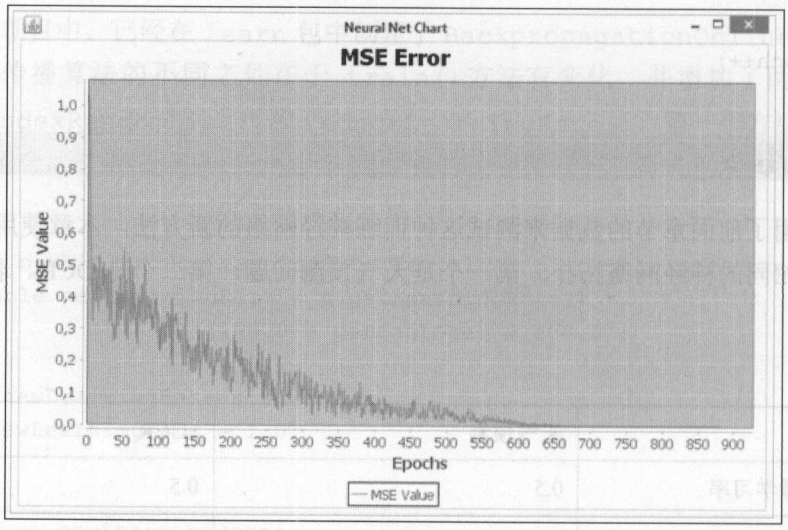


图 9-9

我们还进行了其他实验：用反向传播算法训练神经网络，考虑用在线方法找到学习率。在这两个问题中的 MSE 值都下降了。

天气预测的 MSE 大约为 0.206，而原始的 MSE 是 0.287（第 5 章中得到的），如图 9-10 所示。

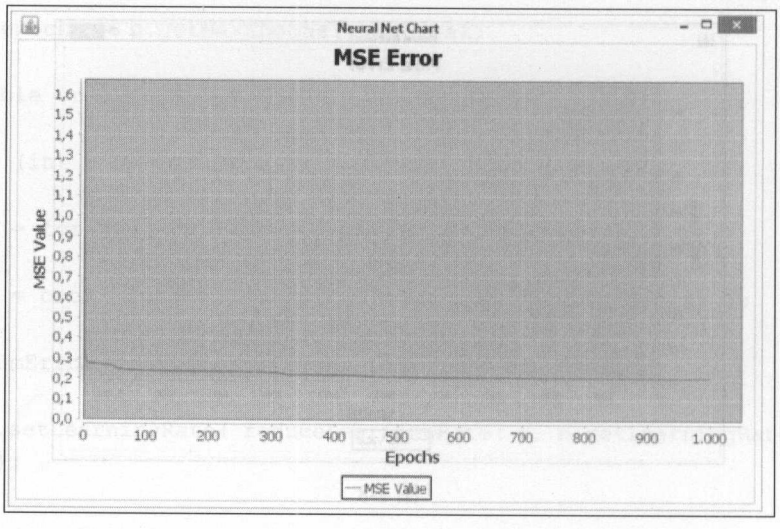


图 9-10

OCR 的 MSE 大约在 8.663E-6，而原始的 MSE 是 0.001（第 8 章中得到的），在图 9-11 中可以看到这一点。

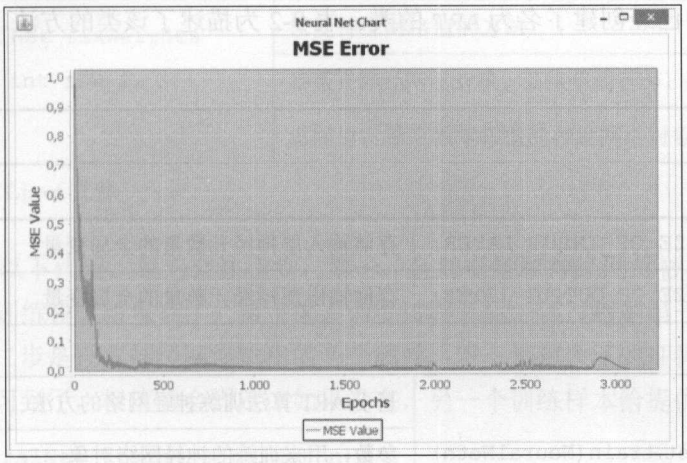


图 9-11

另一个重要的观察是基于以下事实：图 9-11 中所示的训练过程几乎在第 3000 次迭代终止。因此，它比使用相同算法的第 8 章中讨论的训练过程更快更好。

9.5 自适应神经网络

与人类学习相似，神经网络也可以为了不忘记之前的知识而运行。使用传统的神经学习方法，这几乎是不可能的，因为每次训练都涉及替换已有的连接为新连接，从而“忘记”了先前的知识，因此需要通过增加而不是替换它们当前的知识来使神经网络适应新的知识。为了解决这个问题，我们将探讨一种被称为自适应共振理论（adaptive resonance theory，ART）的方法。

9.5.1 自适应共振理论

推动这一理论发展的问题如下：“一个自适应系统如何在保留对某个显著输入的可塑性的同时还保持对无关输入的稳定性？”换句话说：“在学习新信息的同时如何保留之前学习的信息？”

我们了解了在处理模式识别的无监督学习中的竞争学习，其中相似的输入产生了相似的输出或激发了相同的神经元。在 ART 拓扑中，为了提供来自竞争层和输入层的反馈而从神经网络中检索信息时，共振就出现了。所以，当神经网络接收到用来学习的数据，就存在由竞争层和输入层之间的反馈所引起的振荡。当模式完全在神经网络内部发展时，振荡会趋于稳定，这种共振就巩固了存储的模式。

9.5.2 实现

在 som 包中已经创建了名为 ART 的类。表 9-2 为描述了该类的方法。

表 9-2

类名：ART	
属性	
private int SIZE_OF_INPUT_LAYER;	存储输入层神经元数量的全局变量
private int SIZE_OF_OUTPUT_LAYER;	存储输出层神经元数量的全局变量
方法	
public NeuralNet train(NeuralNetn)	基于 ART 算法训练神经网络的方法
	参数：用来训练的神经网络对象
	返回值：训练后的神经网络对象
private void initGlobalVars(NeuralNet n)	初始化全局变量的方法
	参数：神经网络对象
	返回值：-
private NeuralNet initNet(NeuralNet n)	初始化神经网络权值的方法
	参数：神经网络对象
	返回值：有初始化权值的神经网络对象
private int calcWinnerNeuron(NeuralNet n, int row_i, double[][] patterns)	计算优胜神经元的方法
	参数：神经网络对象、训练集行号、训练集模式
	返回值：优胜神经元的索引
private NeuralNet setNetOutput(NeuralNet n, int winnerNeuron)	设置神经元输出的属性
	参数：神经网络对象、优胜神经元索引
	返回值：有输出属性的神经网络对象
private boolean vigilanceTest(NeuralNet n, int row_i)	验证神经网络是否学习的方法
	参数：神经网络对象、训练集行号
	返回值：如果神经网络学习了，返回 True；反之，则返回 False

续表

方法	
private NeuralNet fixWeights (NeuralNet n, int row_i, int winnerNeuron)	恢复神经网络值的方法
	参数：神经网络对象、训练集的行号、获胜神经元的索引
	返回值：带有固定权值的神经网络对象
Java 实现类：ART.java 文件	

训练方法见以下代码。我们会注意到，第一，全局变量和神经网络被初始化了，之后，训练集的个数和训练模式被保存了，接下来，训练过程开始。该过程的第一步是计算优胜神经元的索引；第二步是设置神经网络输出的一个属性。下一步包含了验证神经元是否学习到了知识。如果学习到了，权值就会固定；如果没有，另一个训练样本给提供给神经网络。

```
public NeuralNet train(NeuralNet n){

  this.initGlobalVars( n );

  n = this.initNet( n );

  int rows = n.getTrainSet().length;

  double[][] trainPatterns = n.getTrainSet();

  for (int epoch = 0; epoch < n.getMaxEpochs(); epoch++) {

    for (int row_i = 0; row_i < rows; row_i++) {

      int winnerNeuron = this.calcWinnerNeuron( n, row_i, trainPatterns );

      n = this.setNetOutput( n, winnerNeuron );

      boolean isMatched = this.vigilanceTest( n, row_i );

      if ( isMatched ) {
        n = this.fixWeights(n, row_i, winnerNeuron);

      }

    }
```

```
    }

    return n;

    }
```

9.6 小结

本章讨论了使神经网络更好工作的一些主题。这些技术对用神经网络设计解决方案帮助很大。欢迎读者将此框架应用于任何可以使用神经网络来解决的任务中，以便探索这些结构所具有的增强功效。即便是像选择输入数据这样的简单细节也能影响到整个学习过程，过滤糟糕数据或者去除冗余变量亦是如此。我们用两个实现来论证了一些理论。两个策略（随机在线学习和自适应共振理论）有助于提高性能，这些方法论使神经网络扩展知识并因此去适应一个新的变化的环境。

附录 A
NetBeans 环境搭建

本附录显示了如何设置 NetBeans IDE 开发环境的详细步骤。

下载与安装 NetBeans

提示：

下载和安装 NetBeans 之前，请先确认你已经安装了 java 语言开发工具包（Java Development Kit，JDK），你可以从 `https://www.oracle.com/java/index.html` 下载此工具包。

按以下步骤下载和安装 NetBeans。

1. NetBeans 可以在该项目的网站 `https://netbeans.org/downloads/index.html` 上免费下载（如图 A-1 所示）。请根据你的操作系统选择要下载的安装程序，如下拉列表所示。

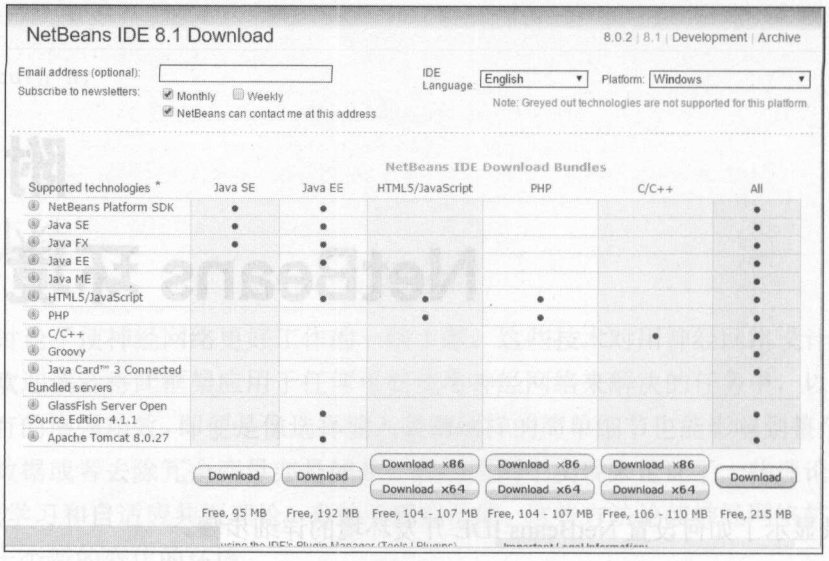


图 A-1

2．你可以从网页显示的版本中进行选择。对于本书中的项目，Java SE 版本非常好，此外，它是最小和最轻的版本。下载应该很快开始，下载任何版本，你都可以单击“download it here”按钮，如图 A-2 所示。

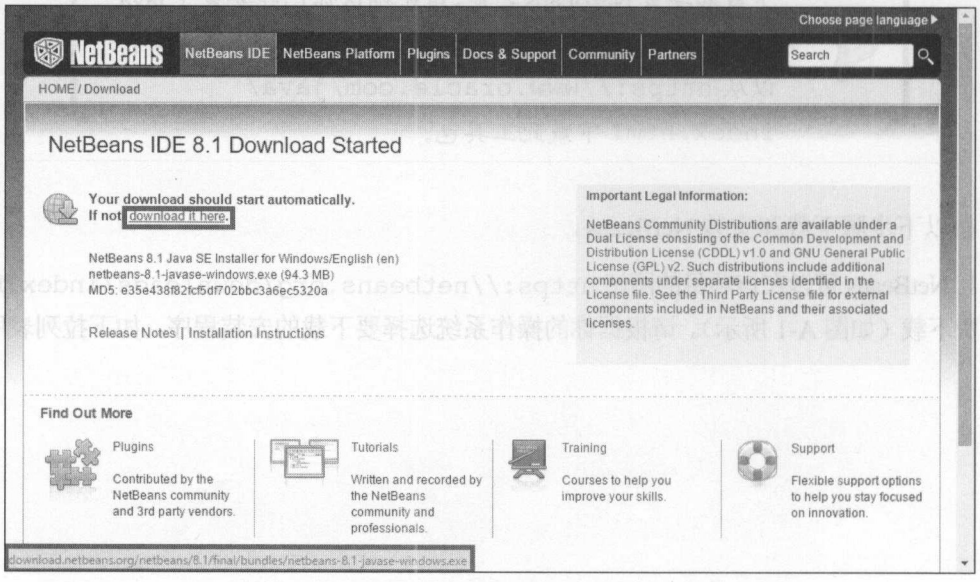


图 A-2

3. 下载后，你应该运行 netbeans- <version> -javase- <your_ os> .exe 可执行文件。此时，出现以下界面，然后可以执行标准安装。在执行安装程序时，它会告诉你安装软件的大小和版本。然后，你可以单击“下一步”（Next）按钮，如图 A-3 所示。

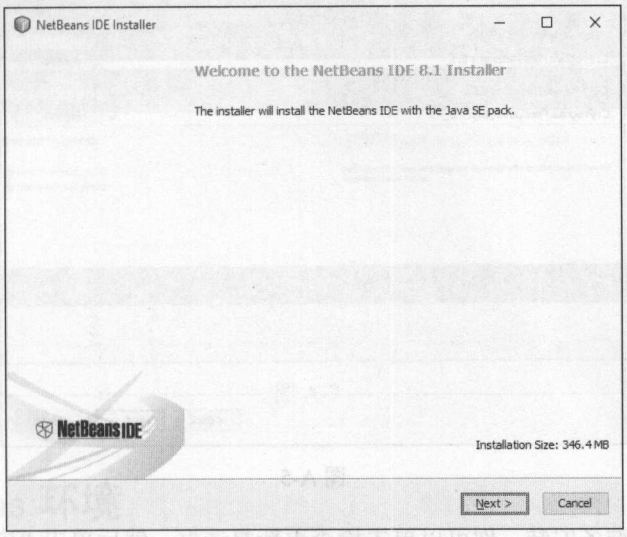


图 A-3

4. 你必须通过选中复选框来接受许可协议，如图 A-4 所示，然后单击“下一步”（Next）按钮。

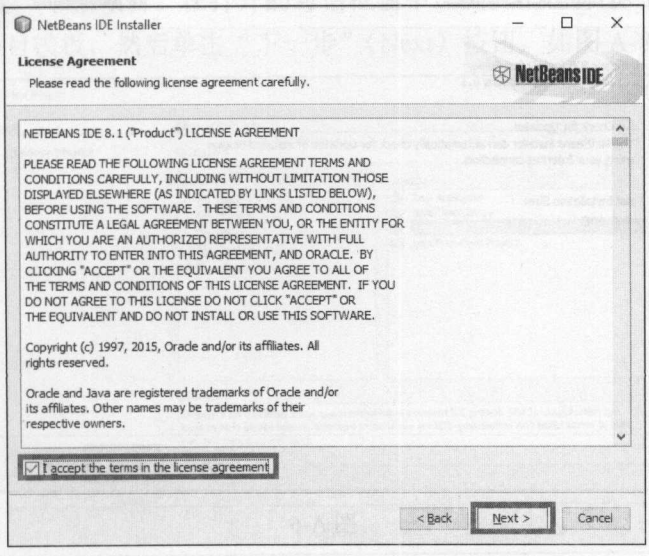


图 A-4

5．然后，你可以选择程序的安装路径，或者可以保留默认路径，如图 A-5 所示。

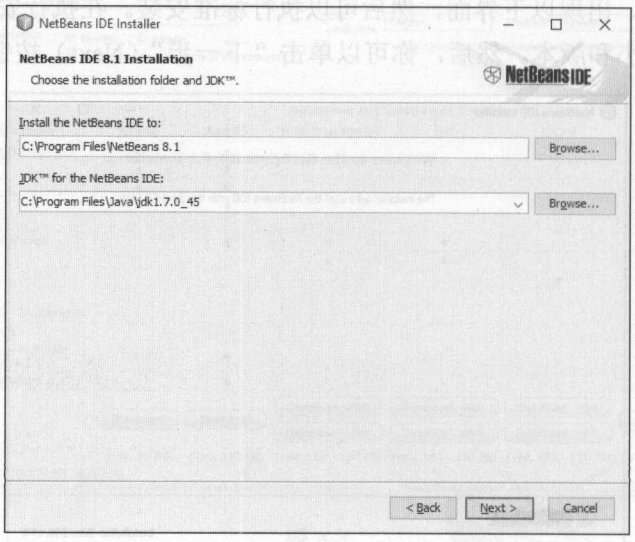


图 A-5

6．现在，我们准备安装。你可以单击检查更新复选框，然后单击“安装”（Install）按钮，如图 A-6 所示。

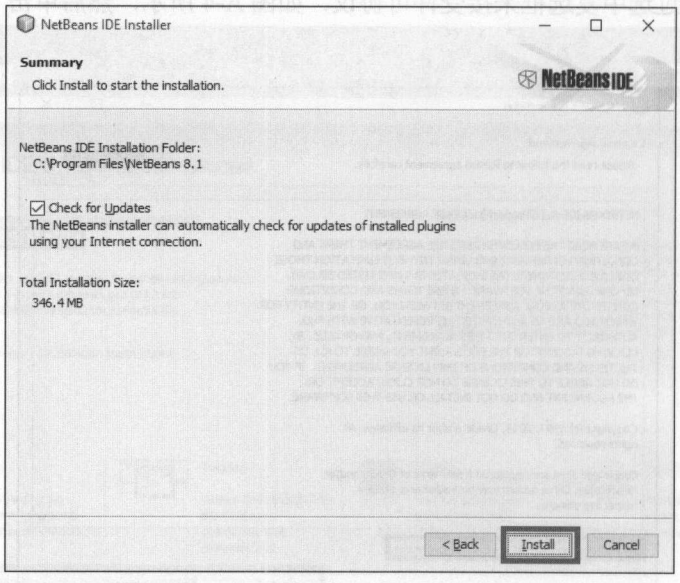


图 A-6

7．安装后，你可以单击桌面上的 NetBeans 图标运行 NetBeans。此时，将打开一个初

始页面，如图 A-7 所示。

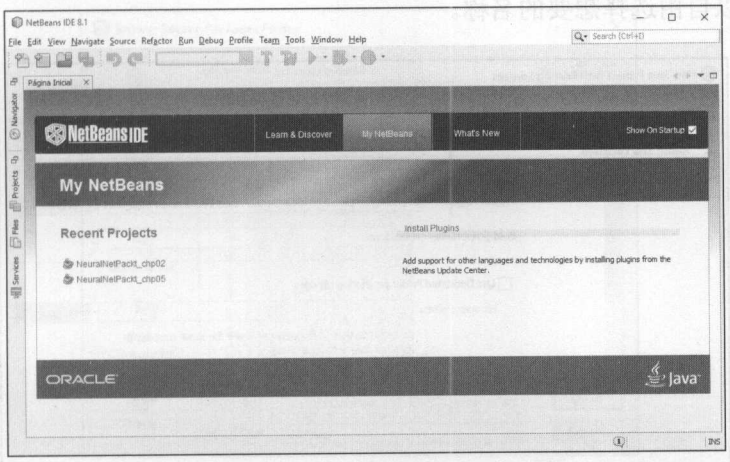


图 A-7

设置 NetBeans 环境

要设置 NetBeans 环境，你需要执行以下步骤。

1．NetBeans 环境已提供了创建和打开新项目的选项。现在，让我们通过选择菜单 File |New Project 来创建一个新项目。在打开的对话框窗中，确保已选择了 **Java Project with Existing Sources** 项目模板，然后单击“下一步”（Next）按钮，如图 A-8 所示。

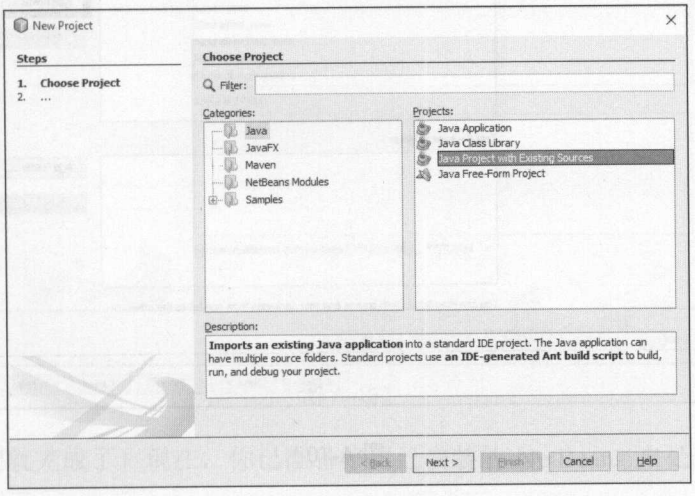


图 A-8

2．然后，你可以为项目选择一个名称（见图 A-9），名称 NeuralNetPackt_ch01 只是一个建议，你可以自由选择想要的名称。

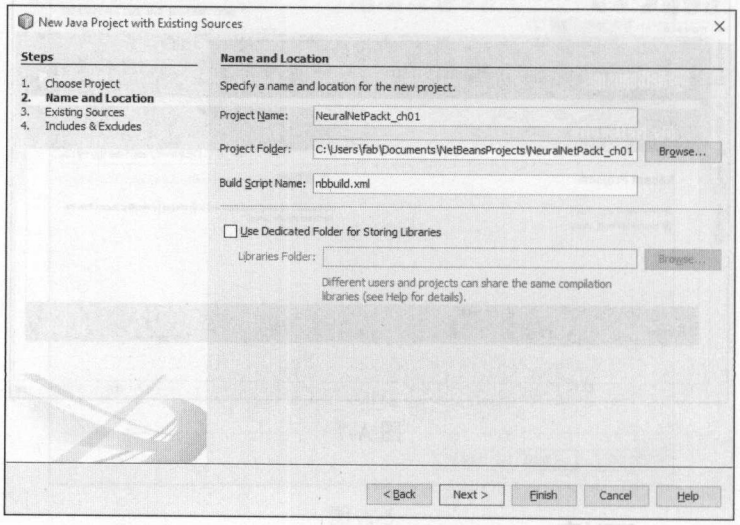


图 A-9

3．在下一个窗口中，你可以选择存储源代码的目录。

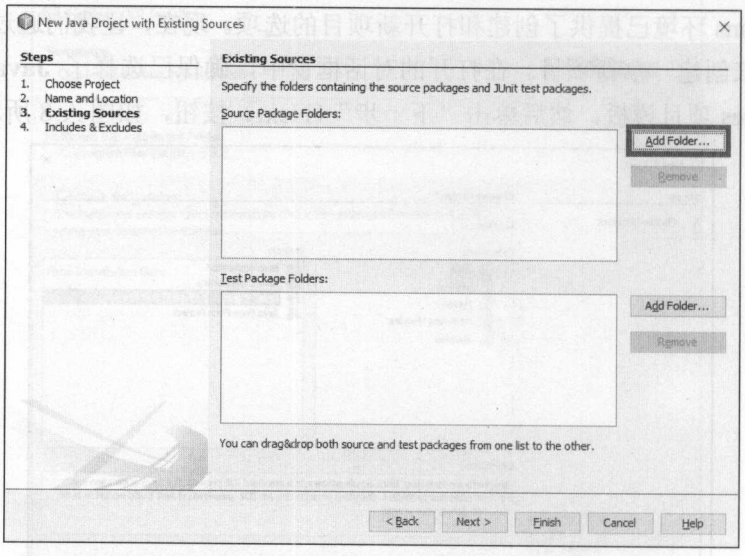


图 A-10

4．在文件选择窗口中，展开已经打开的对话框，浏览存储文件的目录，然后选择文件，

如图 A-11 所示。

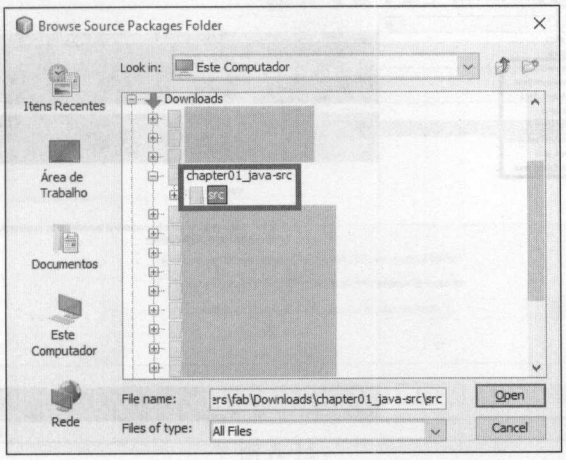


图 A-11

5. 一旦你已经选择文件夹，可以单击“打开”（Open）按钮，然后单击上一级窗口的“下一步”（Next）按钮。现在，显示包含和排除的列表。你可以保持原样，然后单击完成按钮，如图 A-12 所示。

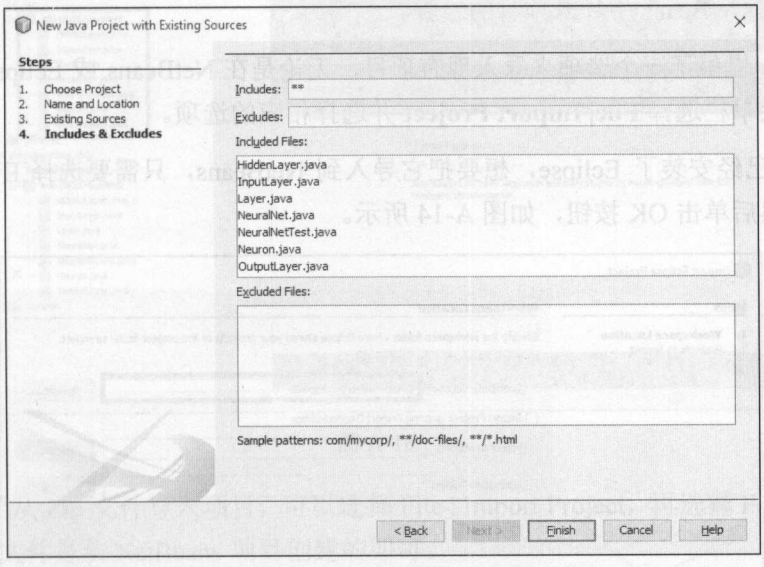


图 A-12

6. 然后我们就完成了！现在，你已准备好用已安装的 NetBeans 来处理每个章节的代码，如图 A-13 所示。

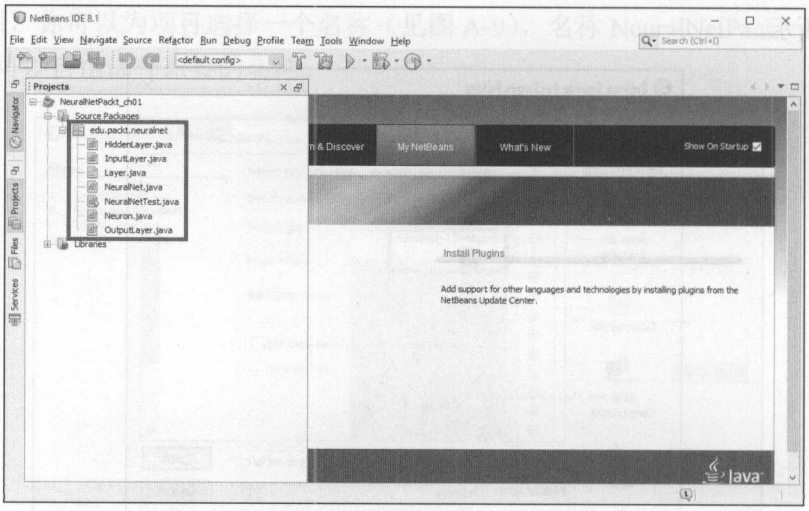


图 A-13

导入项目

以下是在 NetBeans 中导入项目的步骤。

NetBeans 提供了一个选项来导入现有项目，无论是在 NetBeans 或 Eclipse 上创建的。你都可以去菜单栏选择 **File| Import Project** 并选择相应的选项。

1．如果已经安装了 Eclipse，想要把它导入到 NetBeans，只需要选择工作空间所在位置的目录，然后单击 OK 按钮，如图 A-14 所示。

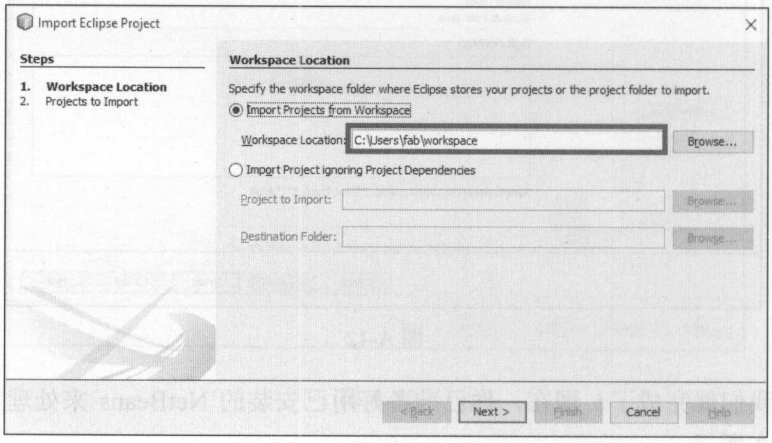


图 A-14

2. 选择想要导入的项目，然后单击 Finish 按钮，如图 A-15 所示。

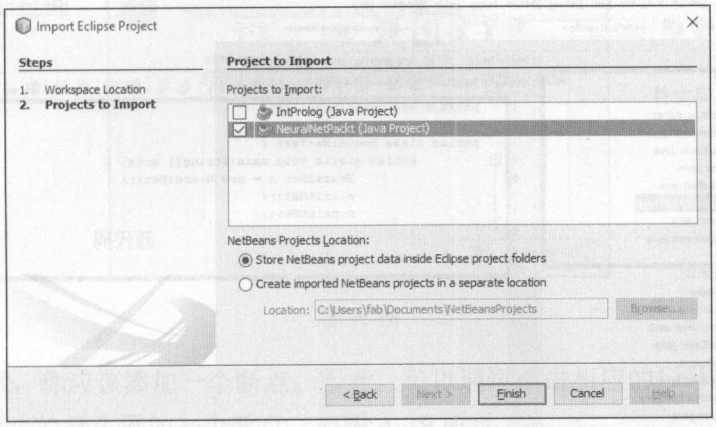


图 A-15

3. 项目成功导入，如图 A-16 所示。

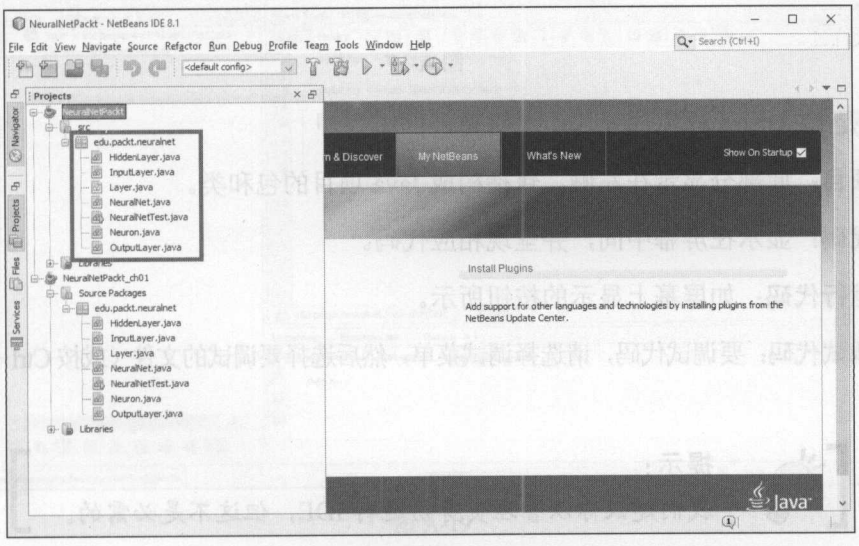


图 A-16

4. 如果要从 Zip 文件导入项目，可以选择 File | Import Project，再选择 From Zip 选项。只需确保 Zip 文件是从 NetBeans 项目创建的即可。

NetBeans 编程和运行代码

在完成所有之前步骤后，你就可以启动 Java 编程。图 A-17 显示了 NetBeans 环境的结构。

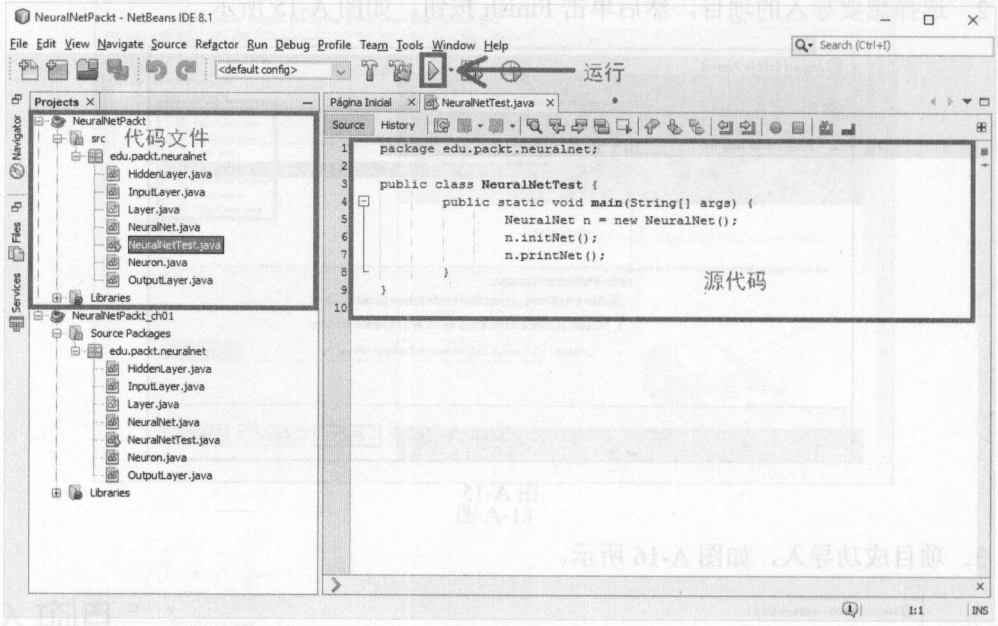


图 A-17

以下是 NetBeans 环境部分的详细信息。

- **项目**：此部分显示在左侧，包括构成 Java 项目的包和类。
- **代码**：显示在屏幕中间，并呈现相应代码。
- **运行代码**：如屏幕上显示的按钮所示。
- **调试代码**：要调试代码，请选择调试菜单，然后选择要调试的文件（或按 Ctrl + Shift + F5 组合键）。

提示：

我们建议你以管理员身份运行 IDE，但这不是必需的。

NetBenas 调试

要在 NetBeans 中调试 Java 程序，只需选择要调试的项目或类文件本身，如图 A-18 所示。

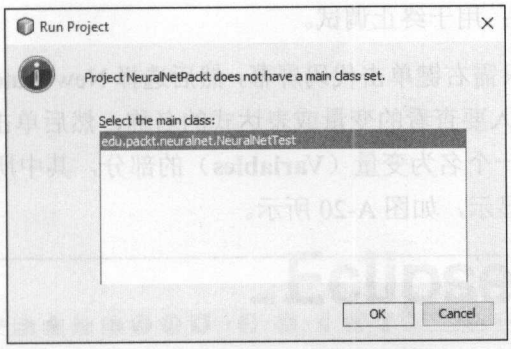


图 A-18

要逐行调试，你应该添加一个断点。因此，可以通过单击相应的行号来放置断点。让我们在 `main` 方法的开头添加一个断点，如图 A-19 所示。

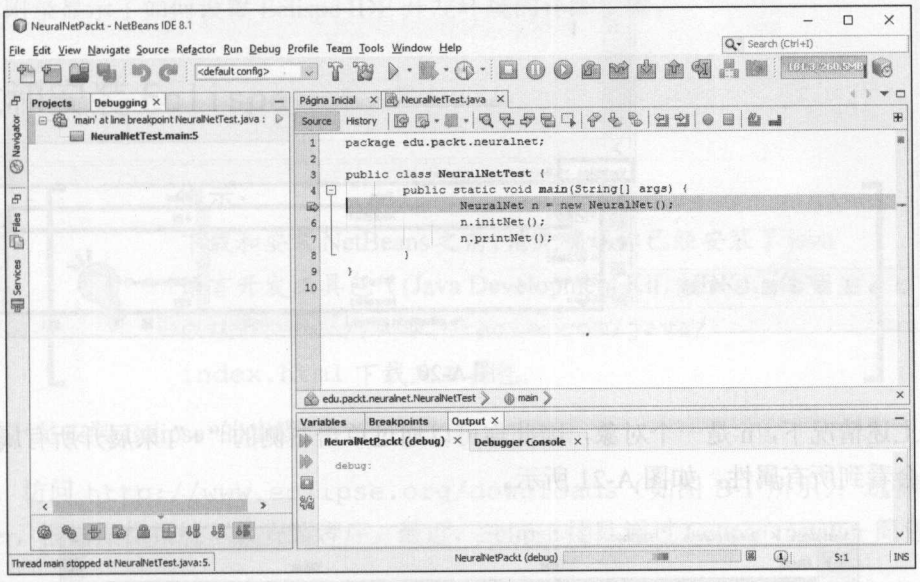


图 A-19

使用以下命令逐行调试源代码。

- F5：用于进入方法。
- F6：用于跳过方法。
- F7：用于返回（从当前方法中跳出，继续往下执行）。
- F8：用于恢复调试。

- Ctrl + F2 组合键：用于终止调试。

要检查变量的值，只需右键单击代码屏幕，然后选择 New Watch 选项（或只需按 Ctrl + Shift + F7 组合键）。插入要查看的变量或表达式的名称，然后单击“确定”（OK）按钮。你可以在屏幕底部看到一个名为变量（**Variables**）的部分，其中所有用户自定义表达式和相关变量均以其当前值显示，如图 A-20 所示。

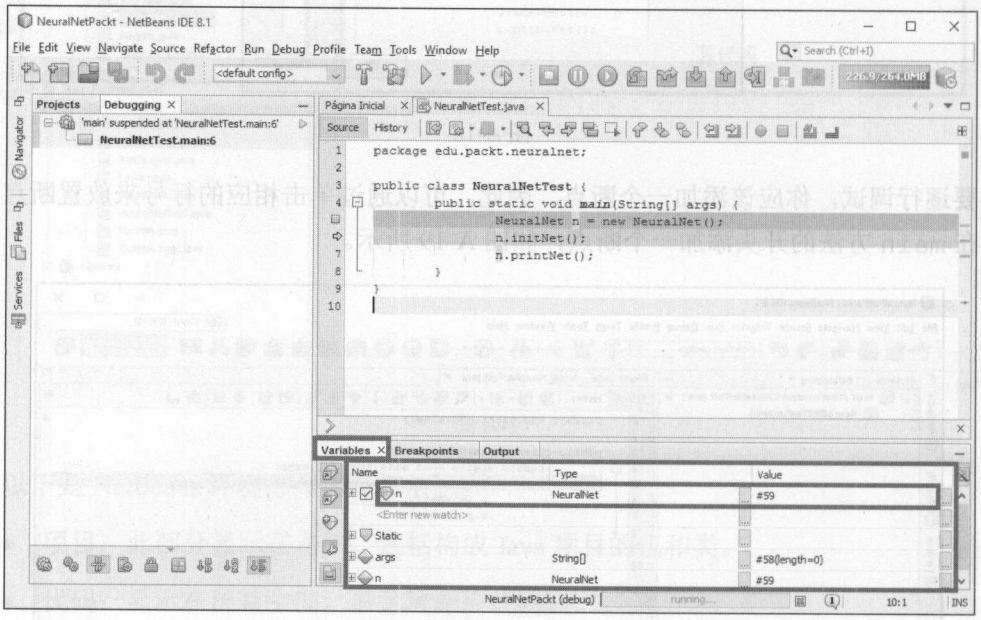


图 A-20

在上述情况下，n 是一个对象，因此你可以通过单击左侧的“+”来展开所有属性。然后你就会看到所有属性，如图 A-21 所示。

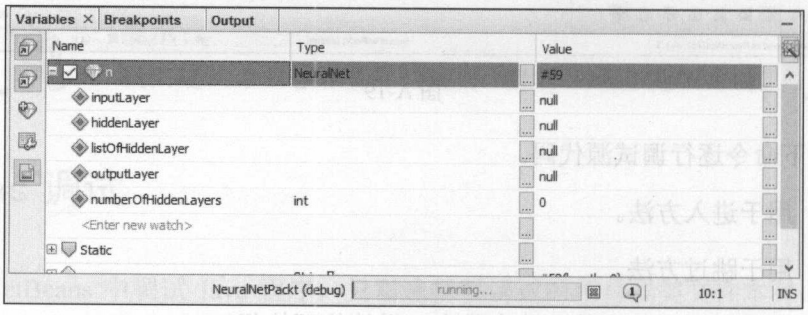


图 A-21

附录 B
Eclipse 环境搭建

本附录显示了如何设置 Eclispe IDE 开发环境的详细步骤。

下载和安装 Eclispe

提示：

下载和安装 NetBeans 之前，请先确认你已经安装了 java 语言开发工具包（(Java Development Kit，JDK)，你可以从 `https://www.oracle.com/java/index.html` 下载此工具包。

以下是安装 Eclipse 的步骤。

1. 访问 `http://www.eclipse.org/downloads`（如图 B-1 所示）；选择 Eclipse Installer，根据操作系统下载安装程序。最近，Eclipse 团队通过 Eclipse Installer 简化了安装过程。

2. 之后会显示图 B-2 中所示的网页。你应该单击“下载”（Download）按钮。自动选择下载的最佳镜像，但如果要选择其他镜像，则可以在页面底部进行选择。

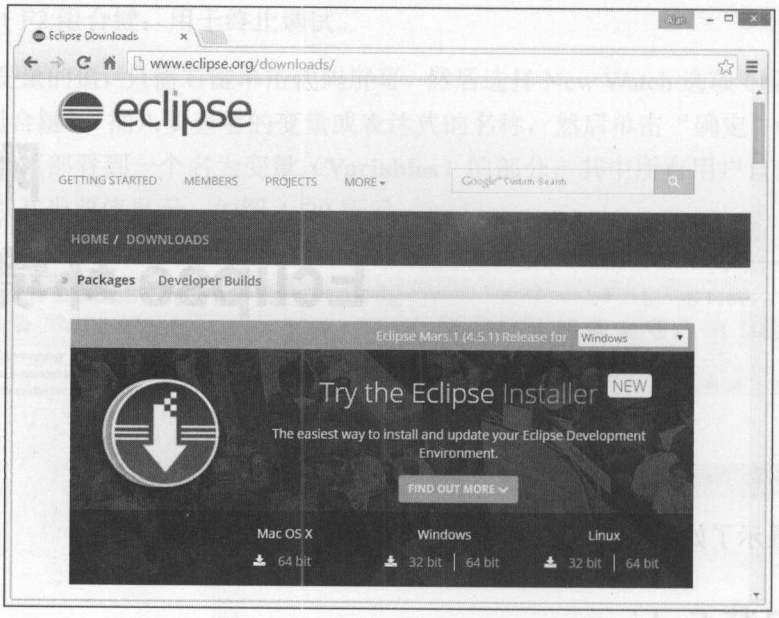


图 B-1

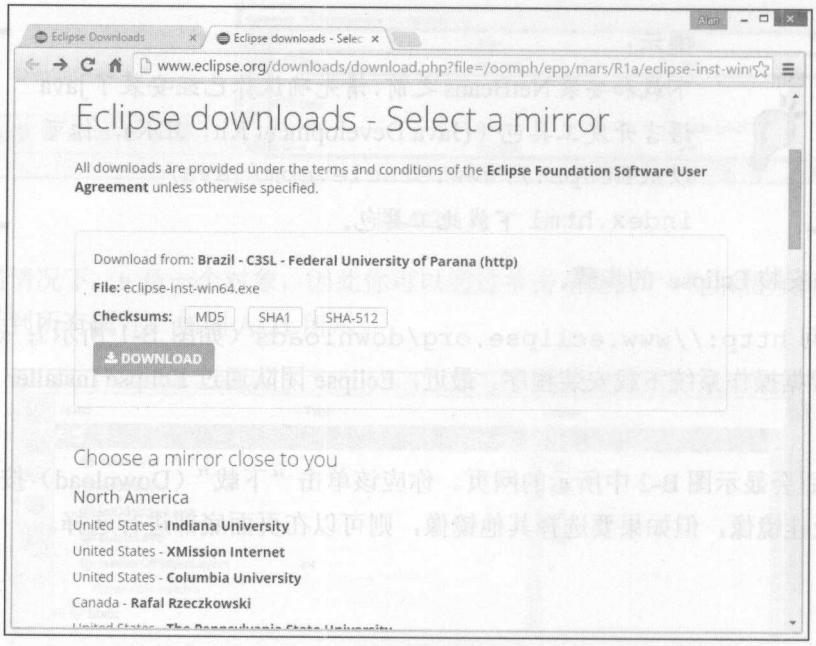


图 B-2

3. 下载后，你应该运行 `eclipse-inst- <your_os> .exe` 可执行文件。之后会

显示以下窗口。由于我们不会开发 Web 应用程序，因此你应该单击第一个选项，即 **Eclipse IDE for Java Developers**，如图 B-3 所示。

图 B-3

4．现在，你应该选择安装文件夹，并决定是否要创建开始菜单条目和桌面快捷方式。然后，单击 INSTALL 按钮，如图 B-4 所示。

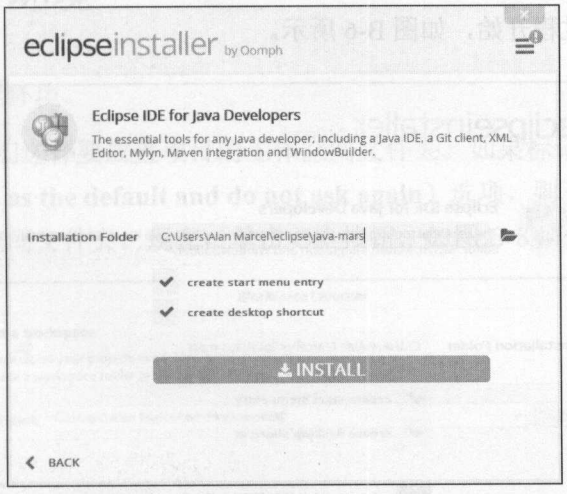


图 B-4

5．你必须通过单击 Accept Now 按钮接受许可，如图 B-5 所示。

图 B-5

6．最后，安装过程开始，如图 B-6 所示。

图 B-6

7．安装后，你可以单击 LAUNCH 按钮运行 Eclipse，如图 B-7 所示。

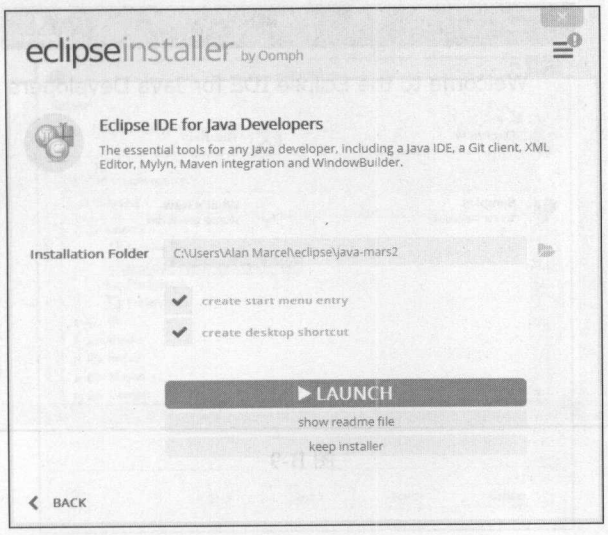


图 B-7

设置 Eclipse 环境

按以下步骤设置环境。

1．下一步，我们选择要放置项目的工作空间文件夹。如果标记为“使用此为默认值，不再询问”（**Use this as the default and do not ask again**）选项，则下次运行 Eclipse 时，不需要再次提示工作空间文件夹。现在，单击 OK 按钮，如图 B-8 所示。

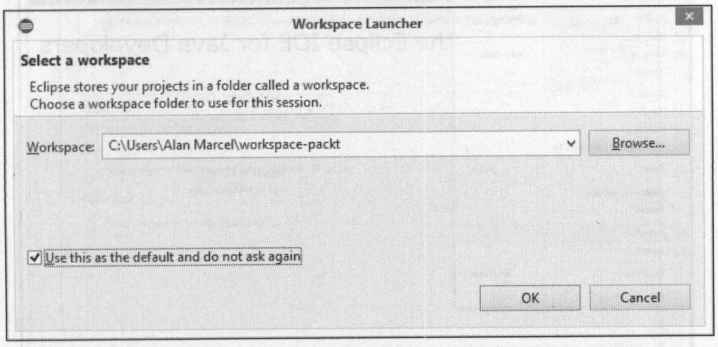


图 B-8

2．显示欢迎界面，如图 B-9 所示，你可以开始 Java 编程了。

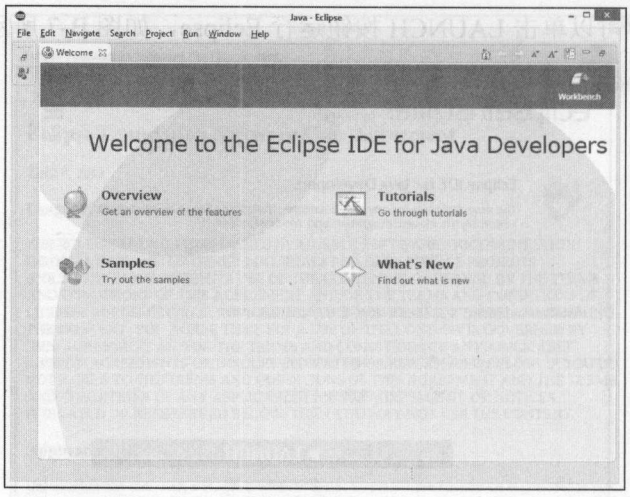


图 B-9

项目导入

提示：

导入项目之前，确保将项目解压到已知的目录中。

以下步骤是导入一个项目到 Eclispe。

1. 要导入已在 Eclipse IDE 中开发的项目，请选择 File | Import 选项，如图 B-10 所示。

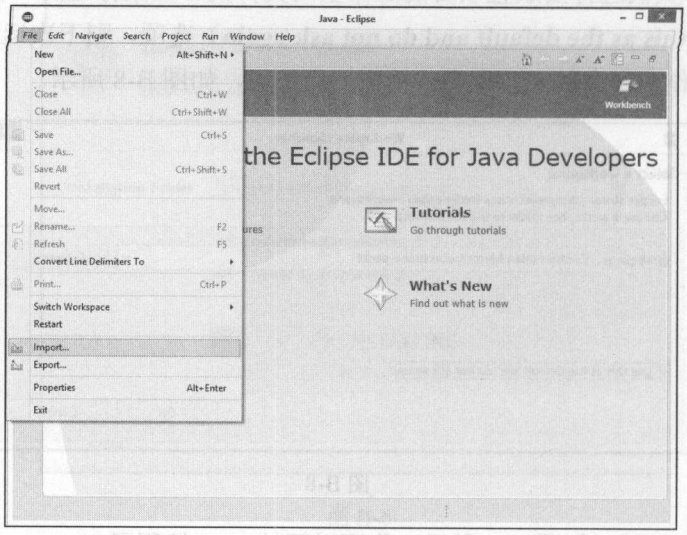


图 B-10

2．显示导入提示后，应展开 General 选项，选择 Existing Projects into Workspace，然后单击“下一步”（Next）按钮，如图 B-11 所示。

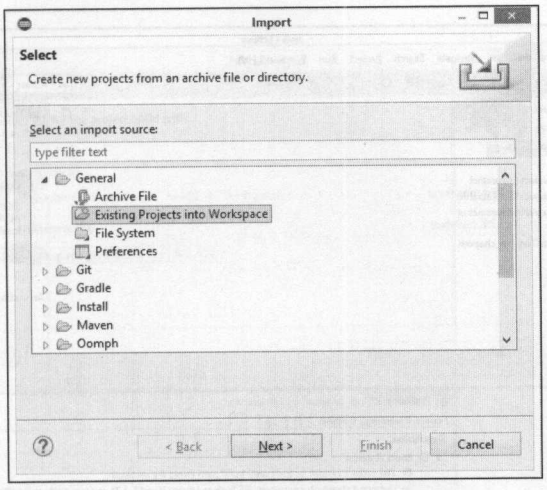


图 B-11

3．现在，单击浏览（Browse...）按钮搜索项目的解压缩目录。之后，不要忘记标记搜索嵌套项目（Search for nested projects）选项，然后单击“完成”（Finish）按钮，如图 B-12 所示。

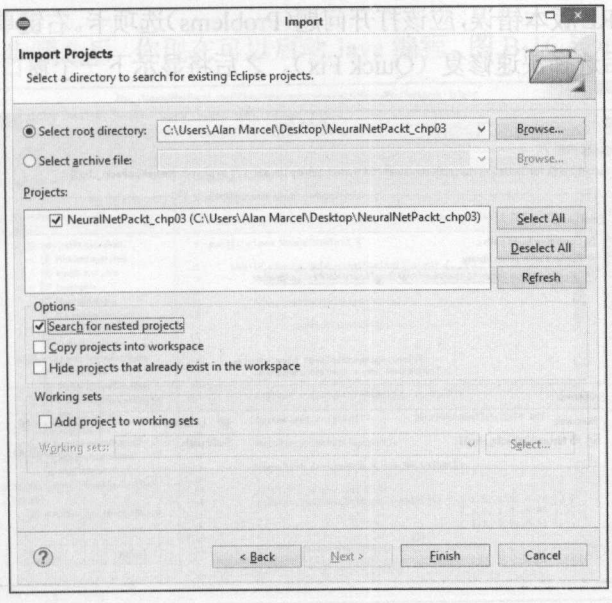


图 B-12

4. 在本步骤中，你应该关闭欢迎界面，你将在 Package Explorer 中看到已导入到 Eclipse 的项目，如图 B-13 所示。

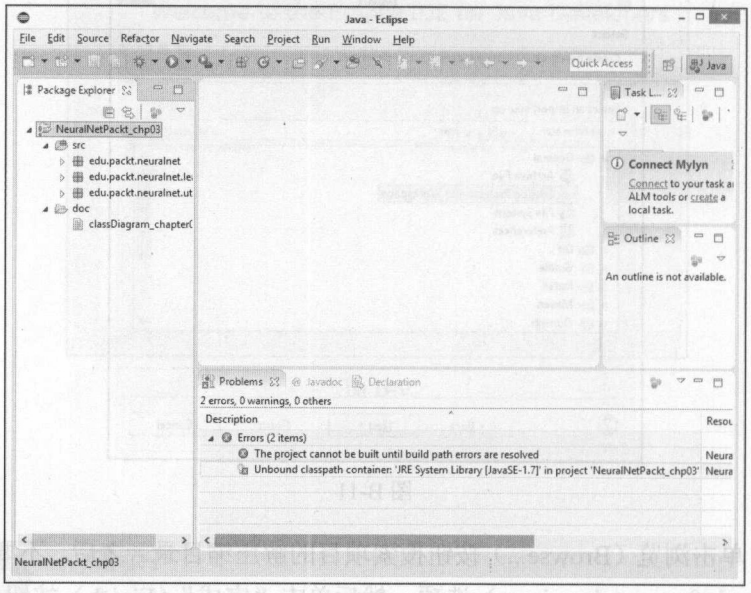


图 B-13

5. 如果你看到 JRE 版本错误，应该打开问题（Problems）选项卡。右键单击 Unbound classpath container...按钮，然后选择快速修复（Quick Fix）。之后将显示下一个窗口，如图 B-14 所示。

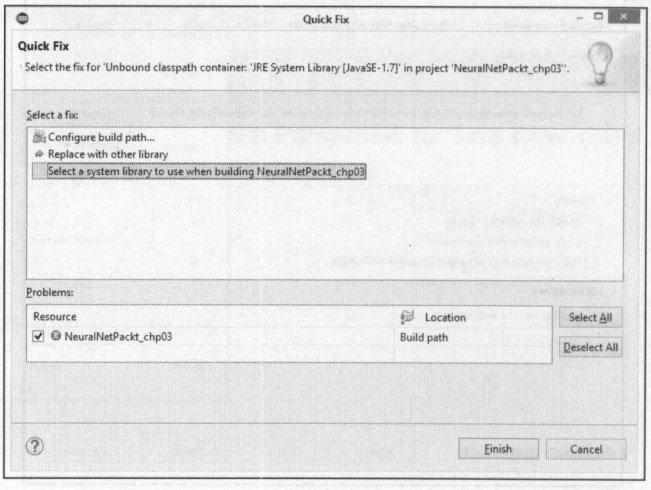


图 B-14

6. 现在，单击 Select a system library to use...选项和“完成”(Finish)按钮。将出现 Edit Library 窗口，你应该选择 Workspace default JRE (jre1.8.0_40)选项，然后单击“完成”（Finish）按钮，如图 B-15 所示。

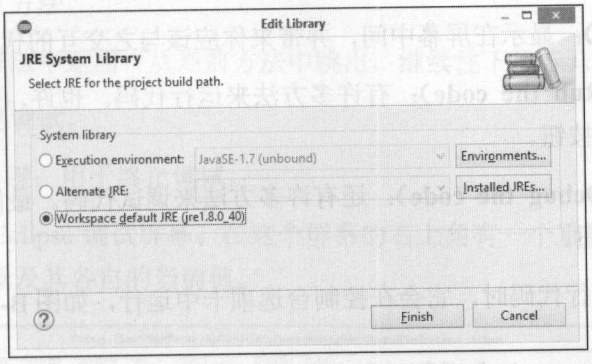


图 B-15

Eclipse IDE 编写和运行代码

提示：

确保以管理员身份运行 Eclipse IDE。

在完成所有上述步骤之后，你现在可以启动 Java 编程。图 B-16 显示了 Eclipse 的结构。

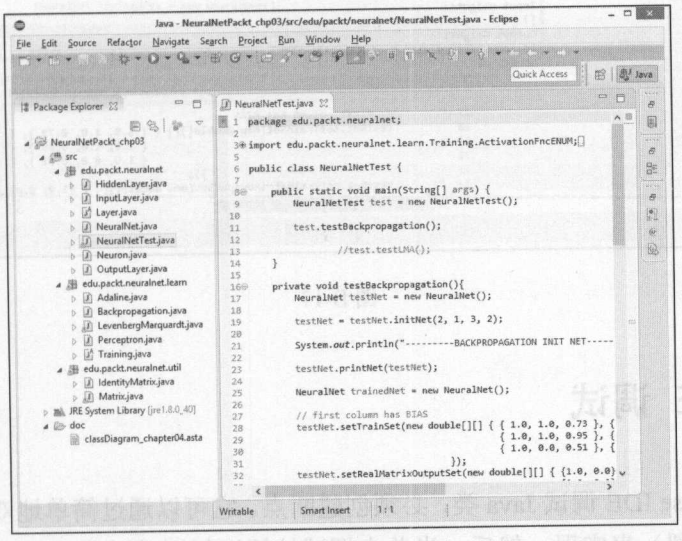


图 B-16

以下是 Eclipse 的 4 个部分。

- **包资源管理器（Package Explorer）**：此部分显示在左侧，包括构成 Java 项目的包和类。
- **代码（Code）**：显示在屏幕中间，并带来你应该与之交互的代码。
- **运行代码（Run the code）**：有许多方法来运行代码。也许，最简单的是由箭头 A 指向的 play 按钮。
- **调试代码（Debug the code）**：还有许多方法来调试代码。最简单的是由箭头 B 指定的 bug 按钮。

当你单击按钮运行代码时，它会在控制台选项卡中运行，如图 B-17 所示。

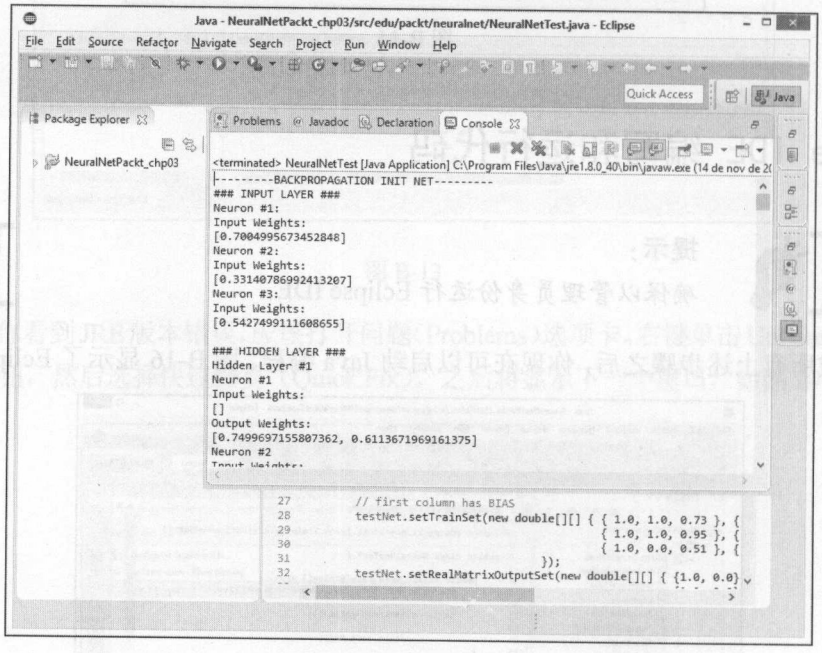


图 B-17

EclipseIDE 调试

要使用 Eclipse IDE 调试 Java 类，必须创建断点。它可以通过简单地双击行号附近（将显示一个蓝色的圆）来实现。然后，当单击调试按钮运行调试过程时，类的执行将立即停

在标记有断点的行上，你可以在键盘上按以下键。

- F5：用于进入方法。
- F6：用于跳过方法。
- F7：用于从方法中返回（从当前方法中跳出，继续往下执行）。
- F8：用于恢复调试。
- Ctrl + F2 组合键：用于终止调试。

图 B-18 显示了 Eclipse 调试屏幕。在这个屏幕的右上角有一个重要的部分，名为变量（Variables），显示变量及其各自的当前值。

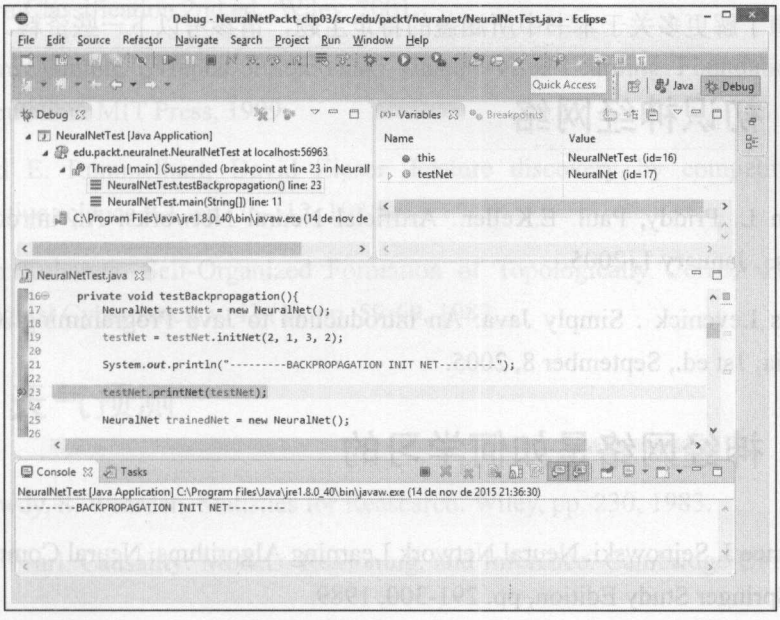


图 B-18

附录 C
参考文献

如果你想了解更多关于本书中所涵盖的特定主题，请参考以下一些资料。

第 1 章　初识神经网络

- Kevin L. Priddy, Paul. E.Keller.. Artificial Neural Networks: An introduction. SPIE Press. . January 1, 2005.
- James Levenick . Simply Java: An introduction to Java Programming. Charles River Media; 1st ed., September 8, 2005.

第 2 章　神经网络是如何学习的

- Terrence J. Sejnowski. Neural Network Learning Algorithms. Neural Computers Volume 41. Springer Study Edition, pp. 291-300, 1989.
- Derrick H. Hguyen, Bernard Widrow. Neural Networks for Self-Learning Control Systems. IEEE Control Systems Magazine, April 1990.

第 3 章　运用感知器

- Simon O. Haykin. Neural Networks and Leaning Machines. Prentice Hall, 3rd ed., November 28, 2008.
- David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams. Learning Representations

by back-propagating errors. Nature v. 323 (6088), pp. 533-536, October 8, 1986.

- K. Levenberg. A Method for the Solution of Certain Non-Linear Problems in Least Squares. Quaterly of Applied Mathematics, vol 2, pp. 164-168, 1944.
- D. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. SIAM Journal on Applied Mathematics, vol 11 (2), pp. 431-441, 1963.

第 4 章　自组织映射

- Richard O Duda, Peter E. Hart, David G. Stork. Unsupervised Learning and Clustering, Pattern Classification 2nd ed.. Wiley, 2001.
- Geoffrey Hinton, Terrence, J. Sejnowski. Unsupervised Learning: Foundations of Neural Computation. MIT Press, 1999.
- David E. Rummelhart, David Zipser. Feature discovery by competitive learning. Cognitive science 9.1, pp. 75-112, 1985.
- Teuvo Kohonen. Self-Organized Formation of Topologically Correct Feature Maps. Biological Cybernetics, v. 43 (1), pp. 59-69, 1982.

第 5 章　天气预测

- S. Dowdy, S. Wearden. Statistics for Reasearch. Wiley, pp. 230, 1983.
- Judea Pearl. Causality: Models, Reasoning, and Inference. Cambridge University Press, 2000.
- Luigi Fortuna, Salvatore Graziani, Alessandro Rizzo, Maria G. Xibilia. Soft Sensors for Monitoring and Control of Industrial Processes. Springer Advances in Industrial Control, 2007.

第 6 章　疾病诊断分类

- Edward I. Altman, Giancarlo Marco, Varetto Franco. Corporate distress diagnosis: Comparison using linear discriminant analysis and neural networks (the Italian

experience). Journal of Banking and Finance v. 18, pp. 505-529, 1994.

- C. M. Bishop. Neural Networks for Pattern Recognition. Oxford University Press, 1995.
- Qeethara K. Al-Shayea. Artificial Neural Networks in Medical Diagnosis. International Journal of Computer Science Issues, Vol. 8, Issue 2, pp. 150-154, March 2011.
- David A. Freedman. Statistical Models: Theory and Practice. Cambrige University Press, 2009.
- Tom Fawcett. An Introduction to ROC Analysis. Pattern Recognition Letters, vol. 27, is. 8 pp. 861-874, 2006.

第7章 客户特征聚类

- Du, K.L. Clustering: A Neural Network Approach. Neural Networks, Vol. 23, Is. 1, pp. 89-107, January 2010.
- J. Park, I.W. Sandberg. Universal Approximation using Radial-Basis-Function Networks. Neural Computation, vol. 3 is. 2, pp. 246-257, 1991.
- Michael E. Wall, Andreas Rechtsteiner, Luis M. Rocha, Singular value decomposition and principal component analysis. A Practical Approach to Microarray Data Analysis, pp. 91-109, 2003.
- Glendon Cross, Wayne Thompson. Understanding your Customer: Segmentation Techniques for Gaining Customer Insight and Predicting Risk in the Telecom Industry. SAS Global Forum, 2008.

第8章 模式识别（OCR 案例）

- Jayanta K. Basu, Debnath Bhattacharyya, Tai-hoon Kim. Use of Artificial Neural Network in Pattern Recognition. International Journal of Software Engineering and Its Applications, Vol. 4, No. 2, April 2010.
- Vivek Shrivastava, Navdeep Sharma. Artificial Neural Network Based Optical Character Recognition. Signal and Image Processing: An International Journal (SIPIJ), Vol. 3, No. 5, October 2012.

第9章 神经网络优化与自适应

- Utrans J. Moody, Rehfuss S., Siegelmann H. Input variable selection for neural networks: application to predicting the U.S. business cycle. Computational Intelligence for Financial Engineering, Proceedings of the IEEE/IAFE, 1995
- Saxén H., Pettersson, F. Method for the selection of inputs and structure of feedforwaed neural networks. Computers and Chemical Enginnering, Vol. 30, Is. 6-7, pp. 1048-1045, May 15, 2006.
- Alan M. F. Souza, Carolina M. Affonso, Fábio M. Soares, Roberto C.L. De Oliveira. Soft Sensor for Fluoridated Alumina Inference in Gas Treatment Centers. Intelligent Data Engineering and Automated Learning 2012, Lecture Notes in Computer Science v. 7435, pp. 294-302, Spinger Verlab Berlin Heidelberg, 2012.
- Jollife. I.T. Principal Component Analysis. 2nd ed. Springer Wiley, 2002.
- Karmin, E.D. A simple procedure for pruning back-propagation trained neural networks. IEEE transactions on Neural Networks, pp. 239-242, June 1990.
- P.E. Gill, W. Murray, M.H. Wright. Practical Optimization. Academic Press: London, 1981.
- Gail A. Carpenter, Stephen Grossberg. Adaptive Resonance Theory. The Handbook of Brain Theory and Neural Networks, 2nd ed., pp. 1-11, 2002

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

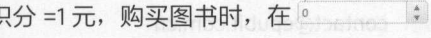
与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享异步社区购书优惠券。

使用方法：注册成为社区用户，在下单购书时输入 S4XC5 使用优惠码，然后点击“使用优惠码”，即可在原折扣基础上享受全单9折优惠。（订单满39元即可使用，本优惠券只可使用一次）

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

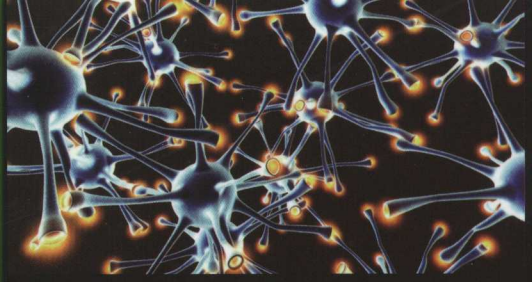
加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ 群：436746675

社区网址：www.epubit.com.cn

投稿 & 咨询：contact@epubit.com.cn



神经网络算法与实现
——基于Java语言

神经网络已成为从大量原始的、看似无关的数据中提取有用数据的强大技术。 Java语言是用于实现神经网络的最合适的工具之一，也是现阶段非常流行的编程语言之一，包含多种有助于开发的API和包，具有“一次编写，随处运行”的可移植性。

本书完整地演示了使用Java开发神经网络的过程，既有非常基础的实例也有高级实例。首先，你将学习神经网络的基础知识、感知机及其特征。然后，你将使用你学到的概念来实现自组织映射网络。此外，你还会了解一些应用，如天气预报、疾病诊断、客户特征分析和光学字符识别（OCR）等。最后，你将学习实时优化和自适应神经网络的方法。

本书所有的示例都提供了说明性的源代码，这些源代码综合了面向对象编程（OOP）概念和神经网络特性，以帮助你更好地学习。

本书的目标读者

本书非常适合任何对神经网络技术感兴趣的开发人员和业余读者阅读，读者只需具备基本的Java编程知识，无需了解神经网络的相关概念。本书将从零开始为读者进行由浅入深的讲解。

通过阅读本书，你将能够：

- 掌握神经网络的知识及用途；
- 运用常见实例开发神经网络；
- 探索和编码最广泛使用的学习算法，让你的神经网络可以从大多数类型的数据中学习知识；
- 发现神经网络的无监督学习过程的力量，提取隐藏在数据背后的内在知识；
- 应用实际示例（如天气预测和模式识别）中生成的代码；
- 了解如何选择最合适的学习参数，以确保应用更高效；
- 选择数据集，将数据集切分为训练集、测试集和验证集，并探索验证策略；
- 了解如何改善和优化神经网络。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

ISBN 978-7-115-46093-6

ISBN 978-7-115-46093-6

定价：59.00 元

分类建议：计算机／人工智能／神经网络
人民邮电出版社网址：www.ptpress.com.cn